热烈祝贺“开发专家之 Sun ONE”系列被
评为电子工业出版社“最佳品牌奖”

开发专家之 Sun ONE 即将转为 Java 开发专家 系列

源自精品 成就理想

出 版 说 明

★ 从“开发专家之 Sun ONE”到“Java 开发专家”

“开发专家之 Sun ONE”系列丛书从诞生之日至今，已经四岁了。在这个系列里面，我们一直努力体现着这么一个理念：用一种较为敏锐的视角来跟踪 IT 技术的发展轨迹，并把可能为广大程序员所希望获得的知识，用图书出版的方式奉献给大家。

在这个系列中，我们陆续出版了约 30 种图书，有《Java 与模式》、《JSP 应用开发详解（第二版）》、《精通 EJB（第三版）》、《Tomcat 与 Java Web 应用开发详解》、《精通 Struts：基于 MVC 的 Java Web 设计与开发》、《JBoss 管理与开发核心技术（第三版）》、《精通 Spring》、《精通 Hibernate：Java 对象持久化技术详解》等一大批读者朋友耳熟能详的作品。很多作品都是在国内没有同类图书的情况下出版的。在这几年的出版工作中，我们时刻感受着市场的风险，也时刻收获着无数读者给我们的认可。

在这个系列中，凝聚了大量资深技术专家的心血。有大家都熟知的阎宏、刘晓华、孙卫琴、罗时飞等，还有一些正在不断腾跃的开发高手。这些非常优秀的国内原创作者们一直都在支持着“开发专家之 Sun ONE”系列的出版工作，在这里，我们要向他们说声：谢谢。

桃李不言，下自成蹊。由于这些年“开发专家之 Sun ONE”在“两个效益”中的杰出表现，电子工业出版社授予这个系列“最佳品牌奖”。

时代不断前进，技术不断变革。为了顺应 Java 领域的技术发展态势，为了赋予这个经典的图书系列更强的生命力，我们将“开发专家之 Sun ONE”升级为“Java 开发专家”。我们将继承原有的出版理念，紧密跟踪技术热点和发展趋势，会聚更多优秀作者，全力奉献更经典的作品。

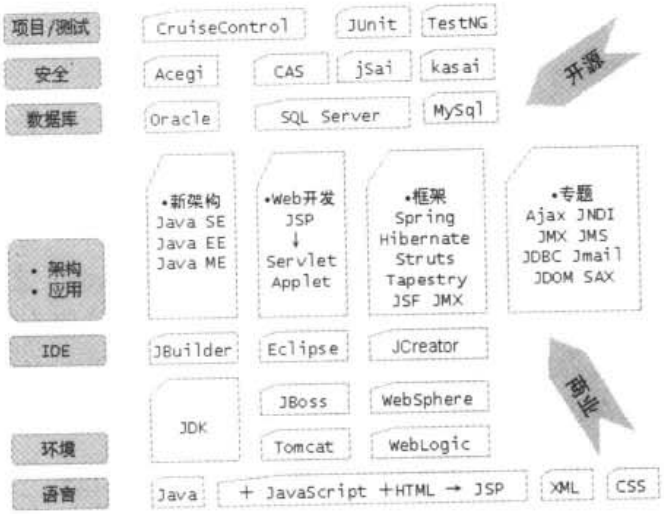
★ 规划你的 Java 开发之路

喜马拉雅山脉的最高峰不断地在温室效应中降低，而 Java 世界的颠峰永远都在技术人员的追求中不断升高。每个人都有不同的路，每个人都有不同的行路方式，不过，往往“到了山顶才发现，错误的路和正确的路就差那么几步！”

身处 Java 洪流中的程序员最累（不过大家都说 Java 程序员薪水最高，呵呵），我们简单整理了一下 Java 领域的相关技术、工具、架构，如下图所示。这个框图中的每一个英文单词（或缩写）都可以写成一本书。Java 领域还有一个特点，那就是商业产品和开源产品层出不穷，潮流不断。相比于其他领域，如.NET，Java 开发更是体现了这句谚语：条条大路通罗马。

罗马只有一个，大路却有多条。看上去，似乎到罗马很容易，反正路多嘛。不过，路多却容易迷失方向。当你在 Java 领域中摸爬滚打几年后，发现自己在无数条道路上走了很久，却不知道罗马何日才能到达，甚至连罗马的方向都不知道，这时你肯定会很失落。

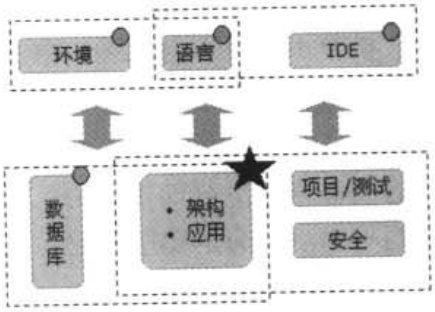
很遗憾，在这个简短的出版说明文章里面，我们无法告诉你每一条连贯的、不费周折的通往罗马的道路该如何走。或许，通过“Java 开发专家”系列中的某本书，你可以找到属于你的正确道路。在一般情况下，我们不会就某一项很窄的话题来单独写一本书，我们还是希望通过我们的一些专业和智慧，尽力把一些相关技术整合起来，用较为简明的方式表达出来，最后由你来选择。



这里有句话与大家共勉：少走弯路，就是捷径！

★ “Java 开发专家”的奉献

犹如在上面那个框图中展现的那样，我们希望在各个层面、各个方向上都能给读者奉献出优秀的图书作品，全面体现技术与应用的结合。从宏观上看，我们会从语言、IDE、环境、数据库、架构与应用、安全、项目与测试等方面进行选择，选出一些读者迫切需要的技术来先行规划。



“Java 开发专家”虽然新蓓初绽，但因其源自盛放的“开发专家之 Sun ONE”系列而根基稳健，两个系列会有一段很长的并行时间，我们会用一种优化的方式来保证读者的顺利选择。无论哪一个系列，必定都有大家喜欢的图书。

在技术上，有着持久化的方法，在学习上，也需要有持久化的精神。
从“开发专家之 Sun ONE”到“Java 开发专家”，希望可以带给你持久化的动力。

联系方式

咨询电话：（010）68134545　　88254160
电子邮件：support@fecit.com.cn
服务网址：http://www.fecit.com.cn　　http://www.fecit.net
通用网址：计算机图书、飞思、飞思教育、飞思科技、FECIT

前　言

随着 Eclipse 加入开源软件的行列，“贡献”这个词越来越被程序员所接受。在使用别人的贡献之余，是不是也想过为 Eclipse 的发展贡献自己的一份力量呢？那么，快快加入 Eclipse 插件开发的行列吧。

在几年前作者开始使用 Eclipse 开发程序的时候，就被 Eclipse 的“短小精悍”所倾倒，后来有幸加入 Eclipse 插件开发的行列，又一次被 Eclipse 的开发模式所征服。但学习 Eclipse 的历程可谓坎坷，在国内能看到的关于 Eclipse 的书籍都是以介绍 Eclipse 使用为主的，在网上能找到的都是一些零散的资料，阅读英文文章和查找 Eclipse 源代码，成了国内插件开发者学习的主要方式。

在国外，Eclipse 的开发已经如火如荼，但在国内这方面还相对落后，中文的资料尤其少之又少。在作者进行 Eclipse 开发之余，某天突然有这样一个想法：“是不是要把自己掌握的一些 Eclipse 知识贡献出来，让国内的 Eclipse 插件开发用户能少走些弯路？”由此开始了本书的写作历程。本人也希望以此为切入点，对国内的 Eclipse 开发做出自己一点小小的贡献，同时在此呼吁国内的程序员更多地参与进来，为国内 Eclipse 的发展尽自己的一份力。

本书概述

本书主要以插件开发为中心，围绕插件开发介绍了 SWT/JFace 的应用、插件扩展点的实现，以及 GEF、EMF 和 RCP 的相关知识。本书的所有讲解范例都能在随书光盘中找到可运行的源代码，并且已经分章节归类。

本书分为 4 部分，第 1 部分为 Eclipse 入门篇，共分为 3 章，介绍了 Eclipse 的一些基本概念、插件的安装使用，以及如何在 Eclipse 中开发 Java 项目。第 2 部分为 SWT/JFace 开发篇，共分 8 章，介绍了 SWT/JFace 开发，其中包括 SWT/JFace 组件、事件和布局的实现。第 3 部分为 Eclipse 插件开发篇，共分 7 章，介绍了 Eclipse 中常用插件扩展点的实现，以及结合 Java 开发，在 Eclipse 的基础上开发插件。第 4 部分为 Eclipse 提高篇，共分 5 章，介绍了 GEF（Graphical Editor Framework）、EMF（Eclipse Modelling Framework），以及 RCP（Rich Client Platform）相关技术，最后在附录中列出了一些 Eclipse 开发的相关资源。

第 1 章：主要介绍 Eclipse 的基本概念，这些概念对插件开发来说是比较重要的。

第 2 章：主要介绍 Eclipse 中 Java 的开发工具（JDT），毕竟 Eclipse 中的开发是基于 Java 的开发。

第 3 章：主要介绍 Eclipse 中的插件安装、诊断和使用，对初学 Eclipse 的用户来说是必备的。

第 4 章：主要介绍 SWT/JFace 的基本概念，以及如何编写 SWT/JFace 的“Hello, World”程序，另外介绍了在 SWT 中如何加入 Swing 的组件。

第 5 章：主要介绍窗口组件的基类“Control”和几个 SWT 组件。

第 6 章：主要介绍 SWT 常用的事件，以及如何实现事件的监听方法。

第 7 章：主要介绍 SWT 的常用布局管理方式，另外还介绍了如何创建自己的布局类。

第 8 章：主要介绍 SWT 中常用的容器组件和其他一些组件。

第 9 章：主要介绍 JFace 的 Viewer 组件，包括 TreeViewer、TableViewer 和 ListViewer。

第 10 章：主要介绍系统对话框和如何自定义用户的对话框。

第 11 章：主要介绍 SWT 的拖放和剪贴板等一些常用的应用。

第 12 章：主要介绍插件和插件扩展点的基本知识。

第 13 章：主要介绍视图扩展点的实现。

第 14 章：主要介绍编辑器扩展点的实现。

第 15 章：主要介绍项目相关扩展点的实现，包括视角扩展点、菜单和工具栏扩展点，以及项目特性等扩展点的实现。

第 16 章：主要介绍帮助扩展点的实现。

第 17 章：主要介绍开发中常使用的几个没分类的扩展点的实现。

第 18 章：主要介绍插件开发中常用到的一些技术，包括构建属性视图、大纲视图和构建编辑器输入（EditorInput）等。

第 19 章：主要介绍 Draw2D 的概念和简单的实现。

第 20 章：主要介绍 GEF 的原理和简单实现。

第 21 章：主要通过一个实例，介绍构建一个完整 GEF 编辑器的过程。

第 22 章：主要介绍 EMF 的概念，以及如何通过 EMF 构建 GEF 模型。

第 23 章：主要介绍 RCP 的相关技术。

第 24 章：主要介绍 Eclipse 一些实现的内幕。

附书光盘

本光盘包含书中所有源代码、使用的插件，共分为两个目录。

1．Plugin 目录

包含本书所用到的相关插件，其中主要有 GEF 插件、EMF 插件、VE 插件和 Eclipse 的汉化包。本书以 Eclipse 软件的英文版进行介绍。另外，解压后可按本书介绍的插件使用方式进行加载。

2．SourceCode 目录

包含本书的所有源程序，每一章的源代码都在此章对应的目录下面。另外，第 4 章至第 11 章的可运行项目在“第 4 章～第 11 章（项目）”目录下面（主要在 com.free.swtjface.demo 项目中），而且将源代码项目直接导入 Eclipse 中即可使用。

联系方式

本书出版后，作者会随时关注 Blog 上的留言，并且将和读者一起探讨 Eclipse 的相关知识。如果有什么好的想法和建议，也可以直接通过 E-mail 联系。

Blog：http://www.blogjava.net/ganshuman

E-mail：ganshuman@126.com

行文至此，终于可以落笔。从选题到完稿，其间经历了多少个不眠之夜！此书编著的经历会对我以后的工作产生重大的影响，这里我要感谢中国软件与技术服务股份有限公司架构部的宋乃辉，谢谢他给予我足够的信任和自由发挥的空间。

由于时间仓促，加上本人能力有限，书中难免会存在不当之处，恳请广大读者批评指正。

编著者

目　录

第 1 部分　Eclipse 入门篇

第 1 章　Eclipse 概述 3
1.1　Eclipse 项目背景 3
1.2　Eclipse 简介 3
1.2.1　开放式体系结构 4
1.2.2　平台体系结构 4
1.3　Eclipse 的主要功能特性 5
1.3.1　文件存放（Workspace） 5
1.3.2　Eclipse 开发环境（Workbench） 6
1.3.3　编译与运行 7
1.3.4　版本管理 7
1.4　工作台 8
1.4.1　资源 8
1.4.2　资源层次结构 8
1.4.3　构建 8
1.5　运行时概述 9
1.5.1　运行时插件模型 9
1.5.2　插件 10
1.5.3　Bundle 10
1.6　开源的选择 11
1.7　小结 12
第 2 章　Eclipse 安装及 Java 开发 13
2.1　Eclipse 开发环境的安装 13
2.1.1　JDK 安装 13
2.1.2　Eclipse 安装 14
2.2　Java 开发工具 JDT 14
2.2.1　视角 14
2.2.2　视图 15
2.2.3　编辑器 16
2.3　在 Eclipse 中开发 Java 程序 16
2.3.1　Java 项目 16
2.3.2　新建“Hello，world”程序 17
2.3.3　运行程序与调试 18
2.3.4　类的构建路径 19
2.3.5　项目构建 20
2.4　其他功能 20
2.4.1　查看源代码 20
2.4.2　重构 21
2.4.3　生成 Javadoc 文档 21
2.5　Eclipse 快捷键 22
2.5.1　编辑相关快捷键 22
2.5.2　查看和定位快捷键 23
2.5.3　调试快捷键 24
2.5.4　常用编辑器快捷键 24
2.5.5　其他快捷键 24
2.6　小结 25
第 3 章　Eclipse 的使用 27
3.1　Eclipse 插件安装 27
3.1.1　拷贝安装 27
3.1.2　links 安装 28
3.1.3　update 安装 29
3.1.4　查看插件 30
3.2　开发简单的 Web 应用 31
3.2.1　安装配置 MyEclipse 31
3.2.2　创建 Web 应用 32
3.2.3　部署并运行 Web 应用 35
3.3　小结 36

第 2 部分　SWTJFace 开发篇

第 4 章　SWT/JFace 入门 39
4.1　SWT/JFace 简介 39
4.1.1　SWT/JFace 概述 39
4.2.2　SWT/JFace 基本特性 40
4.2　SWT“Hello world”程序 41
4.2.1　加入 SWT 依赖的包 41
4.2.2　“Hello world”程序 43
4.2.3　运行“Hello world”程序 43
4.3　JFace“Hello world”程序 44
4.3.1　“Hello world”程序 44
4.3.2　运行“Hello world”程序 45
4.4　在 SWT 中使用 Swing 组件 46
4.4.1　在 SWT 的组件上加入 Swing 的组件 46
4.4.2　程序运行效果 48

4.5 小结 48
第 5 章 SWT 常用组件 49
5.1 Control 类 49
5.1.1 Control 类的继承关系 49
5.1.2 Control 类的常用方法 49
5.2 Button 组件 50
5.2.1 Button 组件实例 50
5.2.2 Button 组件的风格 52
5.2.3 Button 组件的对齐方式 52
5.2.4 加入图片及按钮组合属性 52
5.3 Label 组件 53
5.3.1 Label 组件实例 53
5.3.2 Label 组件的风格和对齐方式 54
5.3.3 加入图片及组合属性 55
5.4 Text 组件 55
5.4.1 Text 组件实例 55
5.4.2 Text 组件的风格 56
5.5 Combo 组件 57
5.5.1 Combo 组件实例 57
5.5.2 Combo 组件的风格 59
5.6 List 组件 59
5.6.1 List 组件实例 59
5.6.2 List 组件的风格 61
5.7 小结 61
第 6 章 SWT 事件 63
6.1 鼠标事件 63
6.1.1 MouseListener 接口 63
6.1.2 MouseMoveListener 接口 64
6.1.3 MouseTrackListener 接口 64
6.1.4 鼠标事件实例 65
6.2 键盘事件 67
6.2.1 KeyListener 接口 67
6.2.2 键盘事件实例 67
6.3 焦点事件 69
6.3.1 FocusListener 接口 69
6.3.2 焦点事件实例 69
6.4 窗口控制事件 71
6.4.1 ControlListener 接口 71
6.4.2 窗口控制事件实例 71
6.5 选择组件事件 72
6.5.1 SelectionListener 接口 73
6.5.2 选择组件事件实例 73
6.6 其他常用事件 74
6.6.1 HelpListener、VerifyListener 和 ModifyListener 监听器的功能 74
6.6.2 HelpListener、VerifyListener 和 ModifyListener 监听器实例 75
6.7 小结 77
第 7 章 SWT 布局管理 79
7.1 概述 79
7.2 FillLayout 布局 80
7.2.1 FillLayout 的风格 80
7.2.2 FillLayout 布局实例 80
7.3 RowLayout 布局 82
7.3.1 RowLayout 的风格 82
7.3.2 RowLayout 布局实例 83
7.4 GridLayout 布局 84
7.4.1 GridLayout 的风格 85
7.4.2 GridData 的相关属性 85
7.4.3 GridLayout 布局实例 87
7.5 FormLayout 布局 91
7.5.1 FormLayout 布局信息 91
7.5.2 FormLayout 布局实例 92
7.6 创建自己的布局 96
7.6.1 Layout 类 96
7.6.2 创建自己的布局类 97
7.7 小结 100
第 8 章 SWT 容器及其他组件 101
8.1 SWT 容器 101
8.1.1 Composite 组件 101
8.1.2 Group 组件 102
8.1.3 TabFolder 和 TabItem 组件 104
8.1.4 SashForm 组件 105
8.2 菜单和工具栏 108
8.2.1 Menu 和 MenuItem 组件 108
8.2.2 ToolBar 和 ToolItem 组件 111
8.2.3 CoolBar 和 CoolItem 组件 113
8.3 其他组件 115
8.3.1 Slider 组件 115
8.3.2 Scale 组件 116
8.3.3 ProgressBar 组件 118
8.4 小结 120

第 9 章 Viewer 组件 121
9.1 TreeViewer 121
9.1.1 TreeViewer 构建步骤 121
9.1.2 内容提供器（ITreeContentProvider） 121
9.1.3 标签提供器（ILabelProvider） 123
9.1.4 TreeViewer 实例 123
9.2 TableViewer 127
9.2.1 TableViewer 构建步骤 127
9.2.2 内容提供器（IStructured ContentProvider） 128
9.2.3 标签提供器（ITableLabelProvider） 128
9.2.4 TableViewer 实例 129
9.3 ListViewer 138
9.3.1 ListViewer 功能 138
9.3.2 ListViewer 实例 138
9.4 小结 142
第 10 章 对话框 143
10.1 消息对话框 143
10.1.1 消息对话框的功能 143
10.1.2 消息对话框实例 144
10.2 输入对话框 147
10.2.1 输入对话框的功能 147
10.2.2 输入对话框实例 147
10.3 进度监视对话框 149
10.3.1 进度监视对话框的功能 149
10.3.2 进度监视对话框实例 150
10.4 自定义对话框 153
10.4.1 带标题区域对话框 153
10.4.2 自定义对话框实例 153
10.5 常用系统对话框 156
10.5.1 打印对话框（PrintDialog） 157
10.5.2 颜色对话框（ColorDialog） 157
10.5.3 字体对话框（FontDialog） 158
10.5.4 文件对话框（FileDialog） 159
10.5.5 目录对话框（DirectoryDialog） 159
10.6 小结 160
第 11 章 SWT 的其他应用 161
11.1 拖放 161
11.1.1 拖放的功能 161
11.1.2 拖放的实现 162
11.2 线程 164
11.2.1 线程同步和异步 164
11.2.2 线程同步和异步的实现 164
11.3 剪贴板 166
11.3.1 剪贴板的功能 166
11.3.2 剪贴板的实现 166
11.4 图形上下文 168
11.4.1 图形上下文的实现 168
11.4.2 自定义组件 170
11.5 鼠标形状（Cursor） 172
11.6 小结 174

第 3 部分 Eclipse 插件开发篇

第 12 章 插件入门 177
12.1 什么是插件 177
12.2 一个简单的插件 178
12.2.1 创建 “Hello，world” 插件 178
12.2.2 调试 “Hello，world” 插件 179
12.2.3 插件打包 180
12.3 插件清单文件介绍 181
12.3.1 OverView：描述了插件的基本信息 182
12.3.2 Dependencies：描述了插件的依赖关系 182
12.3.3 Runtime：指明了运行时的 ClassPath 183
12.3.4 Extensions：指明插件实现的扩展点 183
12.3.5 Extension Points：指明了用户插件提供的扩展点 184
12.4 常用扩展点的描述 185
12.4.1 org.eclipse.ui.editors：代表一个编辑器窗口 185
12.4.2 org.eclipse.ui.views：代表一个视图窗口 185
12.4.3 org.eclipse.ui.perspectives：代表一个扩展视角 187
12.5 小结 187
第 13 章 视图扩展点的实现 189

13.1 Views 扩展点的介绍 189
13.1.1 视图分类 189
13.1.2 常用视图可实现的功能 190
13.2 “Hello，world”视图 190
13.2.1 创建“Hello，world”视图插件 190
13.2.2 “Hello，world”插件扩展点 191
13.2.3 “org.eclipse.ui.views”扩展点的实现类“HelloWorldView” 193
13.2.4 运行“Hello，world”视图 194
13.3 “Tree viewer”视图 195
13.3.1 创建“Tree viewer”视图插件 195
13.3.2 “Tree viewer”插件扩展点 196
13.3.3 “org.eclipse.ui.views”扩展点的实现类“TreeViewTest” 196
13.3.4 运行“Tree viewer”视图 202
13.4 “Image viewer”视图 202
13.4.1 创建“Image viewer”视图插件 202
13.4.2 “org.eclipse.ui.viewActions”扩展点的实现类“PushActionDelegate” 205
13.4.3 “org.eclipse.ui.views”扩展点的实现类“ImageView” 206
13.4.4 运行“Image viewer”视图 207
13.5 Debug 视图的实现 208
13.5.1 查找 Debug 插件 208
13.5.2 导入 Debug 插件 208
13.5.3 Debug 插件中视图的扩展点 209
13.6 小结 211
第 14 章 编辑器扩展点的实现 213
14.1 Editors 扩展点的介绍 213
14.1.1 编辑器分类 213
14.1.2 常用编辑器可实现的功能 213
14.1.3 编辑器和视图的区别 214
14.2 “Hello，world”编辑器 214
14.2.1 “Hello，world”编辑器插件 214
14.2.2 “Hello，world”插件扩展点 215
14.2.3 “org.eclipse.ui.editors”扩展点的实现类“HelloWorldEditor” 216
14.2.4 运行“Hello，world”编辑器 218
14.3 多页签编辑器的实现 219
14.3.1 创建多页签编辑器插件 219
14.3.2 多页签编辑器插件扩展点 220
14.3.3 “org.eclipse.ui.editors”扩展点的实现类“MultiPageEditor” 220
14.3.4 “org.eclipse.ui.editors”扩展点的辅助类“MultiPageEditor Contributor” 225
14.3.5 运行多页签编辑器 227
14.4 简单的文本编辑器 227
14.4.1 创建文本编辑器插件 227
14.4.2 “org.eclipse.ui.editors”扩展点的实现类“FileEditor” 228
14.4.3 运行文本编辑器 231
14.5 小结 231
第 15 章 项目相关扩展点的实现 233
15.1 项目中常用的扩展 233
15.2 视角 233
15.2.1 简介 234
15.2.2 “org.eclipse.ui.perspectiveExtensions”扩展点 234
15.2.3 “org.eclipse.ui.perspectives”扩展点 237
15.3 菜单工具栏 239
15.3.1 “org.eclipse.ui.actionSets”扩展点信息 239
15.3.2 “org.eclipse.ui.actionSets”扩展点的实现类“SampleAction” 240
15.4 项目相关特性 241
15.4.1 项目扩展点 241
15.4.2 “org.eclipse.core.resources.natures”扩展点及实现类 241
15.4.3 “org.eclipse.core.resources.builders”扩展点及实现类 243
15.4.4 “org.eclipse.ui.newWizards”扩展点及实现类 246
15.4.5 “org.eclipse.ui.preferencePages”扩展点及实现类 249

15.5 小结 252
第 16 章 帮助相关扩展点的实现 253
16.1 联机帮助 253
16.1.1 联机帮助扩展点的实现 253
16.1.2 联机帮助文档的结构 254
16.1.3 在联机帮助中在线调用 Java 类 256
16.2 欢迎界面帮助 257
16.2.1 产品扩展点 258
16.2.2 欢迎界面扩展点 258
16.2.3 插件清单文件 259
16.2.4 扩展点实现类 259
16.2.5 调试欢迎界面插件 261
16.3 上下文帮助 261
16.3.1 定义上下文帮助 261
16.3.2 定义弹出菜单和相应对话框 262
16.3.3 运行上下文帮助插件 264
16.4 小结 266
第 17 章 其他扩展点的实现 267
17.1 startup 扩展点的实现 267
17.1.1 “org.eclipse.ui.startup” 扩展点的实现 267
17.1.2 “org.eclipse.ui.startup” 对应的实现类 267
17.1.3 “org.eclipse.ui.startup” 对应插件运行效果 268
17.2 “propertyPages” 扩展点的实现 268
17.2.1 “org.eclipse.ui.propertyPages” 扩展点的实现 268
17.2.2 “org.eclipse.ui.propertyPages” 对应的实现类 269
17.2.3 “org.eclipse.ui.propertyPages” 对应插件运行效果 272
17.3 小结 273
第 18 章 插件开发辅助功能 275
18.1 属性视图 275
18.1.1 IPropertySource 接口 276
18.1.2 属性视图的实现 277
18.2 大纲视图 279
18.2.1 IContentOutlinePage 接口 279
18.2.2 大纲视图的实现 281
18.3 构造 EditorInput 282
18.3.1 IEditorInput 接口 282
18.3.2 IEditorInput 的实现 283
18.4 操纵编辑器 285
18.4.1 查找编辑器 285
18.4.2 打开和关闭编辑器 286
18.5 插件国际化 286
18.5.1 国际化插件的创建 287
18.5.2 类的国际化 287
18.5.3 plugin.xml 文件格式化 289
18.5.4 调试国际化插件 289
18.6 小结 290

第 4 部分 Eclipse 提高篇

第 19 章 Draw2d 简介及实现 293
19.1 Draw2d 简介 293
19.1.1 Draw2d 简介 293
19.1.2 “Hello World” 实现 294
19.2 图形 295
19.2.1 图形（Figure）的功能 295
19.2.2 图形的实现 295
19.3 连线 296
19.3.1 连线（Connection）的功能 297
19.3.2 连线的实现 297
19.4 UML 关系图实现 299
19.4.1 UML 关系图实现的要求 299
19.4.2 CompartmentFigure 实现 300
19.4.3 UMLClassFigure 实现 301
19.4.4 连线及测试类 302
19.5 小结 304
第 20 章 GEF 原理及实现 305
20.1 GEF 概述 305
20.1.1 GEF 简介 305
20.1.2 GEF 工作原理 305
20.2 模型 306
20.2.1 模型的功能 306
20.2.2 模型的实现 306
20.3 视图 311
20.3.1 视图的功能 311
20.3.2 视图的实现 311
20.4 控制器 312

20.4.1 控制器的功能 312
20.4.2 控制器的实现 313
20.5 请求和编辑策略 316
20.5.1 请求和编辑策略（Request and EditPolicies） 316
20.5.2 编辑策略的实现 317
20.6 命令 318
20.6.1 命令（Commands）的功能 318
20.6.2 命令的实现 318
20.7 模型同步 320
20.7.1 模型同步的功能 320
20.7.2 模型同步的实现 320
20.8 选项板 321
20.8.1 选项板（PaletteRoot）的功能 321
20.8.2 选项板的实现 322
20.9 小结 323
第 21 章 GEF 表模型编辑器的实现 325
21.1 程序的软件结构 325
21.1.1 实现功能 325
21.1.2 程序的界面要求 326
21.2 编辑器的实现 326
21.2.1 编辑器的功能 326
21.2.2 编辑器的实现 327
21.3 模型的实现 334
21.3.1 PropertyAwareObject 类 334
21.3.2 Table 类 335
21.3.3 Column 类 339
21.3.4 Schema 类 341
21.3.5 Relationship 类 343
21.4 视图的实现 344
21.4.1 TableFigure 类 344
21.4.2 SchemaFigure 类 346
21.4.3 ColumnsFigure 类 346
21.5 控制器的实现 347
21.5.1 TablePart 的实现 348
21.5.2 RelationshipPart 的实现 357
21.6 EditPartFactory 和 Palette 的实现 359
21.6.1 SchemaEditPartFactory 359
21.6.2 PaletteRoot 360
21.7 EditPolicy 的实现 362
21.7.1 TableNodeEditPolicy（模型连线的编辑策略） 362
21.7.2 TableLayoutEditPolicy（子模型布局的编辑策略） 364
21.7.3 TableContainerEditPolicy（处理模型创建请求的编辑策略） 366
21.7.4 TableEditPolicy（模型编辑的编辑策略） 367
21.7.5 TableDirectEditPolicy（处理直接编辑的编辑策略） 368
21.8 Command 的实现 369
21.8.1 模型连线命令 369
21.8.2 模型拖动命令 372
21.8.3 模型创建命令 373
21.8.4 模型删除命令 374
21.8.5 模型直接编辑命令 377
21.9 大纲的实现 378
21.10 小结 380
第 22 章 EMF 的使用 381
22.1 EMF 的简介 381
22.2 使用 EMF 生成模型 382
22.2.1 定制类关系图 382
22.2.2 生成模型代码 383
22.3 在 GEF 框架中使用 EMF 模型 384
22.3.1 模型 384
22.3.2 EditPart 的修改 384
22.3.3 ElementFactory 的修改 385
22.3.4 处理自定义数据类型 386
22.3.5 保存和载入模型 387
22.4 小结 388
第 23 章 RCP 技术 389
23.1 RCP 的简介 389
23.2 建立一个 RCP 的应用 389
23.2.1 创建 RCP 应用 390
23.2.2 运行 RCP 应用 390
23.2.3 导出 RCP 应用 391
23.3 RCP 的组成部分 393
23.3.1 Plugin.xml 393
23.3.2 应用的入口 Application 395
23.3.3 工作台 ApplicationWorkbench Advisor 395

23.3.4 工作台窗口 ApplicationWorkbenchWindowAdvisor 397
23.3.5 工作台 “Action Bar” ApplicationActionBarAdvisor ...397
23.3.6 应用的视角（perspective） 399
23.4 小结 400
第 24 章 Eclipse 进阶 401
24.1 选择与监听 401
24.1.1 选择事件提供者（Selection Provider） 402
24.1.2 选择服务（Selection Service） 403
24.1.3 选择事件监听器（Selection Listener） 404
24.1.4 实例 405
24.2 命令堆栈 408
24.2.1 命令堆栈（CommandStack） 408
24.2.2 监听器 410
24.3 适配器 410
24.3.1 适配器接口（IAdaptable） 411
24.3.2 适配器的实现 412
24.4 UI 持久化 413
24.4.1 持久化机制 413
24.4.2 UI 持久化与恢复的实现 414
24.5 深入 Workbench 418
24.5.1 Workbench 中实现类的关系 418
24.5.2 Part 的生命周期 419
24.6 资源监听与访问 421
24.6.1 资源监听器 421
24.6.2 资源访问器 422
24.6.3 资源监听的实现 423
24.7 小结 425
附录 A 插件资源列表 427
附录 B 插件开发法则 431
B.1 扩展者法则 431
B.2 促成者法则 431
B.3 发布者法则 432
附录 C 在 Eclipse 中使用 JUnit 433
C.1 JUnit 简介 433
C.2 Eclipse 开发、调试测试程序 434
附录 D Visual Editor 437
D.1 Eclipse Visual Editor 项目介绍 437
D.2 Visual Editor 的内幕 437
D.3 利用 Visual Editor 开发 AWT/Swing 应用程序 438
D.4 Visual Editor 工具 438
D.5 创建和启用用户接口 440
D.6 运行 Visual Class 441
D.7 Visual Editor 对 Eclipse 的贡献 442
附录 E CVS 版本管理 447
E.1 相关术语 447
E.2 工作模式 448
E.3 模块的输入与签出 448
E.4 日常操作 450

Eclipse 插件开发方法与实战

第 1 部分　Eclipse 入门篇

第 1 部分介绍了 Eclipse 的基本概念，另外还介绍了插件的安装和使用，以及如何在 Eclipse 中开发 Java 项目。本部分共分 3 章，第 1 章介绍了 Eclipse 的基本概念，这些概念对插件开发来说是比较重要的。第 2 章介绍了 Eclipse 中 Java 的开发工具（JDT），能让初学者了解如何开发 Java 应用。第 3 章介绍了 Eclipse 中的插件安装、诊断和使用，对初学 Eclipse 的用户来说也是必备的。

本部分力图做到精练，对于刚使用 Eclipse 进行 Java 开发的读者来说，最好能先熟悉一下 Eclipse 的开发环境。另外，在熟悉 Eclipse 之后，希望读者能花些时间掌握快捷键的使用，这必将大大提高用户的开发效率。

第 1 章　Eclipse 概述

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件构建开发环境。本章旨在简要介绍 Eclipse，让读者了解 Eclipse 的基本概念。

本章包括如下主要内容。

- Eclipse 简介
- Eclipse 的主要功能
- 工作台
- 运行时
- 为什么选择 Eclipse

1.1　Eclipse 项目背景

2001 年 11 月 IBM 宣布捐出了价值 4 000 万美金的 Eclipse 项目。Eclipse 是替代 IBM Visual Age for Java（简称 IVJ）的下一代 IDE 开发环境，但它未来的目标不仅仅是成为专门开发 Java 程序的 IDE 环境，根据 Eclipse 的体系结构，通过开发插件，它能扩展到任何语言的开发，甚至能成为图片绘制的工具。之后，IBM 与其他成员公司合作起草了管理条例，成立 Eclipse 基金会（Eclipse Foundation）。在 2004 年初，Eclipse 基金会正式成立。Eclipse 基金会是由很多成员公司组成，独立于 IBM 的组织。

目前，Eclipse 已经提供 C 语言开发的功能插件。更难能可贵的是，Eclipse 是一个开放源代码的项目，任何人都可以下载 Eclipse 的源代码，并且在此基础上开发自己的功能插件。也就是说，只要有人需要，就会有建立在 Eclipse 之上的 COBOL、Perl、Python 等语言的开发插件出现。同时可以通过开发新的插件扩展现有插件的功能，例如在现有的 Java 开发环境中加入 Tomcat 服务器插件。Eclipse 插件可以无限扩展，而且有着统一的外观、操作和系统资源管理，这也正是 Eclipse 的潜力所在。

虽然目前 Eclipse 项目还没有最后完成，但从已有的版本中已经能领略到 Eclipse 的设计主导思想和主要功能特点。更为重要的是，如果能参加到 Eclipse 项目的开发中或者阅读它的开放源代码，这对广大程序员来说无疑是一个千载难逢的提高编程水平的好机会。Eclipse 计划提供多个平台的版本，像 Windows、Linux、Solaris、HP-UX 和 AIX，本书只介绍 Windows 版本。

1.2　Eclipse 简介

Eclipse 是开放源代码的项目，读者可以到http://www.eclipse.org/去免费下载 Eclipse 的最新版本。一般 Eclipse 会提供几个下载版本：Release、Stable Build、Integration Build 和 Nightly

Build，建议下载 Release 或 Stable 版本。本书将以 Eclipse 3.1.1 版本做介绍。

Eclipse 是一个已经完全设计好的平台，用于构建集成 Web 和应用程序开发工具。平台本身不会提供大量的最终用户功能，平台的价值在于它的促进作用：根据插件模型来快速开发集成功能部件。

Eclipse 提供了公共用户界面（UI）模型来使用工具。它被设计为在多个操作系统上运行，同时提供与每个底层操作系统的强大集成。插件可以为 Eclipse 可移植 API 设计程序，并在任何受支持的操作系统上不加改变地运行。

Eclipse 的核心是动态发现、装入和运行插件这样一种体系结构。Eclipse 平台处理查找和运行正确代码的数理逻辑，平台用户界面提供标准的用户导航模型。于是每个插件可以专注于执行少量的任务，有哪些类型的任务、定义、测试、制作动画、发布、编译、调试和图解等，只要用户能想象得到的就会应有尽有。

1.2.1 开放式体系结构

Eclipse 平台定义了一个开放式体系结构，以使使每个插件开发小组专注于他们擅长的领域。让数据库专家构建后端，让可用性专家构建最终用户工具。如果平台设计好了，就可以添加重要的新功能部件和集成级别，而不会影响其他工具。

从最终用户的角度来看，Eclipse 平台使用公共工作台的模型来集成工具。可以使用严格定义的称为扩展点的挂钩来将用户开发的工具插入到工作台中。

平台本身是内置在插件层中的，每个插件定义下层插件的扩展点的扩展，同时定义它们自己的扩展以进行进一步的定制。此扩展模型允许插件开发者向基本工具平台添加各种功能。每个工具的部件（例如文件和其他数据）是由公共平台资源模型来协调的。

平台为用户提供了使用工具的常见方法，并对它们利用插件创建的资源提供集成管理。插件开发者也因此体系结构而受益。平台负责管理不同运行时环境的复杂事务，例如不同操作系统或工作组服务器环境。插件开发者可以专注于完成他们的特定任务，而不用担心这些集成问题。

1.2.2 平台体系结构

对于 Eclipse 插件架构，一个很好的比喻是 Web 浏览器，它也有插件，Eclipse 插件和浏览器相似，都是要解决扩展性问题。对于一个 Web 浏览器，如果用户想既能浏览多媒体动画，又想浏览 PDF 文件，那么用户就需要两个不同的插件，才能保证两种内容都能在 Web 浏览器中显示。Eclipse 也一样，通过它的插件模式用户可以安装不同的插件来进行 HTML 的编辑、数据库的设计、Java 开发、C++开发等，所有这些开发工作完全在一个 Eclipse 安装平台上进行。而且 Eclipse 模型所能做的远超过了这些，理解 Eclipse 插件的关键是首先要理解 Eclipse 扩展点，正是这个扩展点帮用户在 Eclipse 中添加新功能。

当 Eclipse 插件实现了一个扩展点，就创建了一个扩展，此外，扩展还可以创建自己的扩展点。这种插件模式的扩展和扩展点是递归的，而且被证明是非常灵活的。事实上，Eclipse 本身是一个核心的运行时绑定了一些在插件之上构建的插件，换句话说，Eclipse 本身就是一系列的插件。这样随着使用 Eclipse 构建 Eclipse 的经验累积，这种插件模式就变得日渐成熟。

Eclipse 平台是围绕插件概念构建的。插件是向系统提供功能的代码和数据的结构化 Bundle。功能可能通过以下形式提供：代码库（带有公用 API 的 Java 类）、平台扩展，甚至是文档。插件可定义扩展点，即经过严格定义，其他插件可在其中添加功能的位置。

平台中的每个子系统本身是由实现某些关键功能的一组插件构建的。某些插件将可视功能添加至使用扩展模型的平台。其他插件提供可用于实现系统扩展的类库。

Eclipse SDK（Software Developer Kit）包括基本平台及对插件开发特别有用的两个主要工具，Java 开发工具（JDT）和插件开发者环境（PDE）。Java 开发工具（JDT）实现功能完整的 Java 开发环境，插件开发者环境（PDE）添加了专门的工具以使插件和扩展的开发能够顺利进行。这些工具不仅有用，而且还提供许多示例，用来说明如何把插件添加到平台中，如图 1-1 所示。

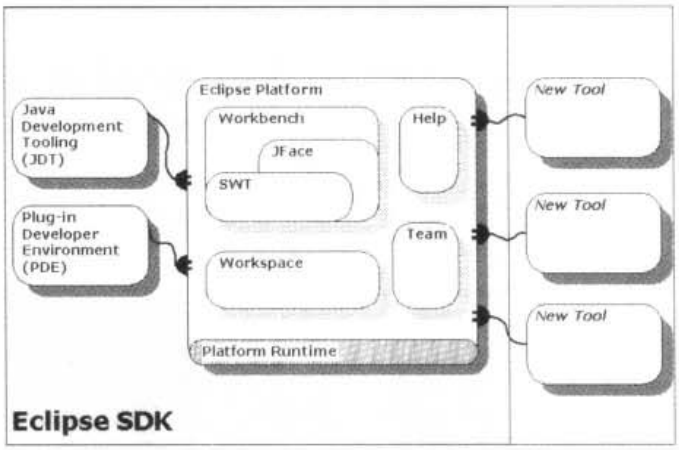


图 1-1 Eclipse 平台体系结构

1.3 Eclipse 的主要功能特性

Eclipse 是在 IVJ 的基础上开发的，它主要包括如下几个方面的功能：文件存放、开发环境、编译与运行、版本管理等。

1.3.1 文件存放（Workspace）

安装 Eclipse 之后，可以在安装路径的下一层路径中创建一个 Workspace（工作区）文件夹（也可以指定磁盘中的其他位置）。每当在 Eclipse 中新生成一个项目，默认情况下都会在 Workspace 中产生和项目同名的文件夹以存放该项目所用到的全部文件。用户可以用 Windows 资源管理器直接访问或维护这些文件。

在 Eclipse 中，Workspace 的概念被简单地描述成可以通过插件访问的资源容器。工作区是终端用户和 Eclipse 平台之间交互的枢纽。终端用户能够在工作区中创建项目并处理其中的内容。工作区本身作为一个目录存在于文件系统中，并且一个 Workspace 在同一时间只局限于一个 Eclipse 实例使用。工作区还包含一个 .metadata 目录，用来持久存储诸如插件状态这样的私有信息。

目前将已有的文件加入到一个项目中有 3 种方式。第一种是用 IDE 的“File”菜单中的“Import”功能将文件导入到项目中。这也是推荐的方式。第二种是从 Windows 的资源管理器中直接拖动文件到项目中。第三种就是直接将文件拷贝到项目文件夹中，然后在 Eclipse 的资源浏览窗口中选择项目或文件夹，并执行本地刷新功能（Refresh）。需要说明的一点是，项目文件夹可以放在计算机的任何位置，并且可以在 Eclipse 中用新建项目的方法将项目路径指定到已经存在的项目文件夹，然后在 Eclipse 中刷新即可。

Workspace 中的项目越多，达到开发系统所能处理项目临界点的机会就越大。一般来说，用户可以创建多个工作区，通过 Workspace 管理不同项目资源。用户可以通过从 Eclipse 中选择“File”→“Switch Workspace”切换工作区，而不必重新启动 Eclipse。

1.3.2 Eclipse 开发环境（Workbench）

Eclipse 开发环境被称为 Workbench，它主要由 3 个部分组成，即视角（Perspective，也称为透视图）、编辑窗口（Editor）和观察窗口（View，也叫视图）。在下面的介绍中，希望读者能知道 Java 视角、Java 包浏览窗口、资源视图和资源浏览窗口等的区别（其实最主要的也就是要区别视角和窗口），以免在进一步的阅读中产生混淆。图 1-2 是它们之间的关系结构略图。

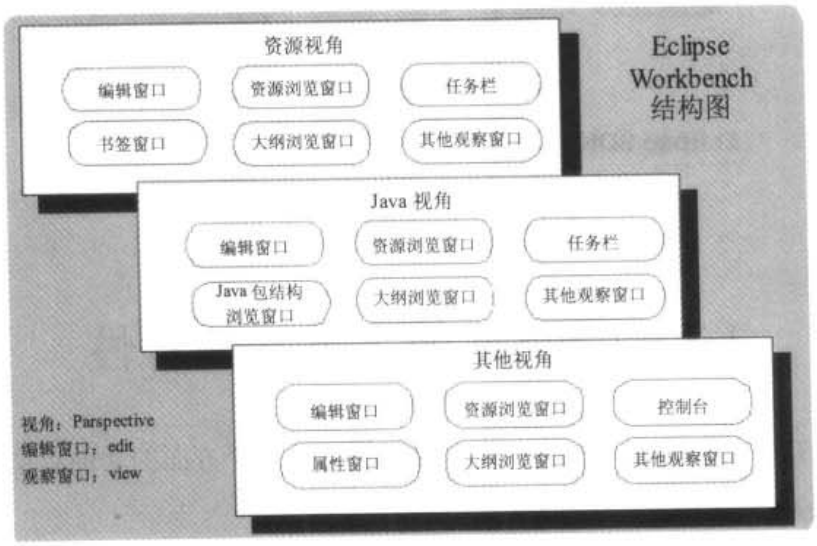


图 1-2 Eclipse Workbench 结构图

从图 1-2 中可以看出 Workbench 包含多个视角，而每个视角又包含不同的窗口。由于每个编辑窗口有很大的共性，而且默认情况它们都在同一区域中显示，因此我们只在每个视角中标出一个编辑窗口。

下面首先介绍编辑窗口。所有文件的显示和编辑都包含在编辑窗口里。默认情况下打开的多个文件是以标签（Tag）方式在同一个窗口中排列的，可以用拖动方式将这些文件排列成各种布局。方法是拖动某一个文件的标签（tag）到编辑窗口的边框，当光标有相应的变化时再释放。

当文件被加入到项目中后，在资源浏览或 Java 包浏览窗口双击文件，Eclipse 会试图打开这个文件，其中 Eclipse 内嵌的编辑器能默认打开一些文件，如*.java、*.txt、*.class 等。如果是其他类型的文件，Eclipse 会调用操作系统相应的默认编辑器打开，如 word 文档、*PDF* 文件等。同时 Eclipse 也可以用指定的编辑器打开相应的文件。

其次要介绍的是观察窗口（除编辑器之外的窗口都是观察窗口），它配合编辑窗口并提供了多种的相关信息和浏览方式。常用的观察窗口有资源浏览窗口（Navigator）、Java 包浏览窗口（Packages）、控制台（Console）和任务栏（Task）等。

浏览窗口和 Java 浏览窗口是观察窗口核心部分。前者和 Windows 的浏览器差不多，能浏览项目文件夹中的所有文件，后者用来浏览项目中的 Java 包、包中的类、类中的变量和方法等信息。在 Java 浏览窗口中可以通过鼠标右键菜单中的“Open Type Hierarchy”命令打开层次浏览窗口（Hierarchy）。这个窗口非常实用，它能非常清晰地查看类的层次结构。类中的编译出错信息可以在任务窗口中查到，同时它也可以成为名副其实的任务窗口：向其中添加新的任务描述信息，来跟踪项目的进度。控制台则主要用来显示程序的输出信息。在调试程序的时候，会有更丰富的观察窗口来帮助程序员进行调试，如变量值查看窗口、断点窗口等。

观察窗口是任何 IDE 开发环境的核心，用好观察窗口也就是用好 IDE 开发环境。Eclipse 提供了丰富的观察窗口，能真正用好这些窗口恐怕得经过一段时间的磨练。

最后介绍视角。一个视角包括一个或多个编辑窗口和观察窗口。视角可以自定义每个视图中包含的观察窗口种类，也可以自定义一个新视角。这些功能都被包括在“Perspective”菜单中。在 Eclipse 的 Java 开发环境中提供了几种视角，如资源视角（Resource Perspective，它也是第一次启动 Eclipse 时的默认视角）、Java 视角（Java Perspective）、调试视角（Debug Perspective）、团队视角（Team Perspective）等。每一种视图都对应不同种类的观察窗口。

1.3.3 编译与运行

在 Eclipse 中调试功能非常强大，有多种跟踪方式、断点设置和变量值查看窗口等调试功能。在 Java 视图中，工具栏中有两个按钮，分别用来进行调试和运行。通常我们需要用的是“Java Applicantion”方式。在这种方式下，如果当前位置是包含 main()方法的 Java 程序，单击“调试/运行”按钮就会立即开始执行“调试/运行”功能。如果当前位置是在包或项目上，Eclipse 会搜索出当前位置所包含的所有可执行程序，然后由程序员自己选择运行哪一个。

在目前 Eclipse 的 Release 和 Stable 版本中默认安装了插件开发环境（Plug-in Development Environment，即 PDE，它本身也是一个插件），此时系统除了“Java Applicantion”运行方式，还有“Eclipse Application”，当用 PDE 开发插件的时候会用到这种运行方式。另外，当用户想运行 SWT 程序和 Junit 程序时，可以选择运行 SWT 程序和 Junit 程序。

1.3.4 版本管理

Eclipse 的版本管理可以分为个人（或称为本地）和团队两种。Eclipse 提供了强大的个人版本管理机制，每一次被保存的更改都可以得到恢复，而且可以精确到每一个方法的版本恢复。操作也十分方便，在任何一个能看到所要操作文件的观察窗口中，例如资源浏览窗口，选中该文件，单击鼠标右键，选择“Compare with”或“Replace with”，如果是恢复已经被删除的方法则可以选择“Previous from local history”，之后相应的本地历史记录就会显示出来，按照用户的需求找到相应的版本就可以了。强大的个人版本管理功能为程序员提供了更多的信心，只管编下去，任何不小心的错误都可以恢复，在 Eclipse 下开发，是有“后悔药”的！

Eclipse 默认为版本管理工具 CVS 提供接口，可以非常方便地连接到 CVS 服务器上。通

过 CVS 版本管理，Eclipse 为团队开发提供良好的环境。要连接 CVS 服务器需要先打开团队视角（Team Perspective），然后在 Reponsitories 观察窗口中单击鼠标右键并选择新建（New），在打开的对话框中可以填入要连接的 CVS 库所需要的信息，操作非常简单。

1.4 工 作 台

工作台（Workbench）的目标是通过为创建、管理和导航工作空间资源提供无缝工具集成的功能。每个工作台窗口都包含一个或多个视角。视角则包含视图和编辑器，并且控制出现在某些菜单栏和工具栏中的内容。在任何给定时间，桌面上都可以存在多个工作台窗口（即在用户看来，Eclipse 可运行多次）。

1.4.1 资源

资源是存在于“工作台”中的项目、文件夹和文件的统称。导航视图提供了资源的分层视图并允许用户打开它们以进行编辑。其他工具可以以不同的方式显示和处理这些资源。

“工作台”中有 3 种基本资源类型。

- 文件：与用户在文件系统中看到的文件类似。
- 文件夹：与文件系统中的目录类似。在“工作台”中，文件夹包含在项目或其他文件夹中。文件夹可包含文件和其他文件夹。
- 项目：包含文件夹和文件。项目用于构建、版本管理、共享和资源组织。与文件夹相似，项目映射至文件系统中的目录。（当创建某个项目时，将在文件系统中为它指定位置。）

另外，项目的状态为打开或关闭。关闭项目后，就不能在“工作台”中更改它。关闭项目的资源将不会出现在“工作台”中，但这些资源仍会驻留在本地文件系统上。关闭的项目需要较少的内存。由于在构建期间不会检查这些项目，所以关闭项目可以缩短构建时间。当项目处于打开状态时，可以更改该项目的结构，且用户可看到有关内容。

1.4.2 资源层次结构

资源是按层次结构存储和显示在“工作台”中的。下面描述的是引用以分层结构存储和显示的资源时所使用的术语。

- 根：“工作台”内容的最上面一层（在文件系统中）。
- 父资源：包含另一资源的任何资源，只有项目和文件夹才可为父资源。
- 子资源：包含在另一资源中的任何资源，只有文件和文件夹才可为子资源。

资源层次结构显示在“Navigator”视图中，该视图是“Resource”视角中的一个默认视图。

1.4.3 构建

构建是一个从现有资源派生新资源和/或更新现有资源的过程。在“工作台”中，不同的构建器是对不同类型的项目调用的。例如，当对 Java 项目触发构建时，Java 构建器将每一个

Java 源文件（.java 文件）转换为一个或多个可执行类文件（.class 文件）。构建器通常强制执行某些域的约束。例如，Web 链接构建器可以更新指向其名称/位置发生更改的文件链接。

有两种构建类型。

（1）增量式构建：增量式构建利用先前构建的状态，以上次构建信息，增量构建更改了的资源。这样只对修改后的内容进行构建，大大提高了构建的效率，例如修改了 Java 文件后通常都是增量构建（Build Automatically）。

（2）完全构建：完全构建将废弃任何问题和先前构建状态。完全构建将根据已配置的构建器域规则变换所有资源。例如，希望重新编辑整个 Java 项目，选择菜单“Project”→“Clean”后将会重新编辑整个 Java 项目。

可对特定的一组项目或对作为一个整体的工作空间执行增量式构建和完全构建。不能构建特定的文件和文件夹。有两种可执行构建的方式。

- 自动构建是在保存资源时构建的。自动构建总是增量式构建并且总是对整个工作空间进行操作。可以配置首选项（“Window”→“Preferences”→“General”→“Workspace”）以在资源修改时自动构建。
- 手工构建是在显式选择菜单项或按等价的快捷键时启动的。手工构建可以是完全构建或增量式构建，并且可以对项目的集合或者整个工作空间进行操作。

1.5 运行时概述

Eclipse 运行时定义插件（org.eclipse.osgi 和 org.eclipse.core.runtime），其他所有插件都依赖于它们。运行时负责定义插件的结构及它们包含的实现详细信息（Bundle 和类装入器）。运行时还负责查找和执行主要的 Eclipse 应用程序及维护插件的注册表，它们的扩展和扩展点。

运行时还提供了一批实用程序（如记录、调试跟踪选项、适配器、首选项库和并行基础结构）。当然，作为最小的内核，运行时插件将坚实地驻留在插件堆的底部，成为 Eclipse 的强大基础。

1.5.1 运行时插件模型

当用户启动使用 Eclipse 开发的应用程序时，平台运行时引擎就会启动。运行时实现平台使用的基本插件模型和基础结构。它保持跟踪所有已安装的插件和它们提供的功能。

插件是一种结构化组件，它将代码或文档添加至系统并以结构化方式描述它。插件可以定义扩展点，即其他插件可以扩展严格定义的功能点。当插件向扩展点添加实现时，我们就认为它向平台添加了扩展。这些扩展和扩展点是在插件的清单文件（plugin.xml）中声明的。定义扩展点与定义其他任何 API 都很相似。唯一的区别就是扩展点是使用 XML 而不是使用代码特征符来声明的。类似地，客户机插件使用 XML 来向系统描述它的特定扩展。

运行时的总体目标是最终用户不应该为已安装但未使用的插件消耗内存或性能。平台扩展模型的说明性质允许运行时引擎确定哪些扩展点和扩展已经由插件提供了但是以前还没有运行过。这样，可以安装许多插件，但是将不会激活它们，直到根据用户的活动请求了插件所提供的功能为止。这是提供灵活可靠平台的一个重要特征。

支持插件的机制是通过使用OSGi框架实现的。从这一点上来说，插件是与 OSGi Bundle 相同的。Bundle 及其相关联的类指定并实现 Java 类装入的过程、先决条件管理和 Bundle 的生命周期。

1.5.2 插件

Eclipse 最有魅力的地方就是它的插件体系结构。在这个体系中重要的概念是扩展点（extension points），也就是为插件提供的接口。每一个插件都是在现有的扩展点上开发，并可能还留有自己的扩展点，以便在这个插件上继续开发。

Plugin 类表示正在平台中运行的插件。通过它可以很方便地将插件生命周期方面的内容和总体语义集中起来。插件可以实现其生命周期启动和停止方面的特殊功能。每个生命周期方法都包括对 BundleContext（它可以提供更多信息）的引用。

生命周期的启动部分是值得特别讨论的。有关插件的信息可以从插件的清单文件中获得，而不必运行任何插件的代码。通常，工作台中的某些用户操作会导致需要启动插件的一系列事件。从实现角度来看，只有在需要装入包含在插件中的类时，才会启动该插件。

start 方法可以很方便地实现插件的初始化和注册行为。但是，认识到可以在许多不同的情况下启动插件是很重要的。像获取修饰对象图标一样简单的操作就可以导致装入插件的某一个类，这样就会启动插件。过早进行初始化会导致在需要装入插件的代码和数据之前很早就装入它们。因此，一定要仔细查看插件的初始化任务并考虑启动时执行初始化的替代方法，启动插件有如下原则。

- 如果可以快速执行注册活动（例如注册监听器或启动后台线程），则可以在插件启动期间执行它们。但是，如果注册活动具有副作用（例如将初始化大型数据结构或者执行不相关的操作），则将这些操作作为访问插件数据的一部分来触发是比较合理的。
- 当第一次访问数据时，最好是延迟进行数据的初始化，而不是在启动代码中自动进行初始化。这样可确保在真正需要大型数据结构之前不会构建它们。

1.5.3 Bundle

生命周期管理是 OSGi“Bundle”和平台“插件”汇合点的管理。当插件启动时，会为该插件提供对BundleContext（从它可以获得与插件相关的信息）的引用。BundleContext还可以用来查明系统中的其他 Bundle（即插件）。

BundleContext.getBundles()可以用来获得系统中的所有 Bundle 集合。可以注册 BundleEvent 的侦听器，以便插件知道何时在其生命周期状态中另一个 Bundle 发生了更改。有关更多信息，请参阅联机帮助的 BundleContext 和 BundleEvent。

BundleActivator 接口定义了 Plugin 中实现的启动和停止行为。尽管 Plugin 类很容易实现此功能，但是插件开发者可以完全自由地在适合于插件设计的任何类中实现 BundleActivator 的接口（默认生成的插件类就实现了此接口）。实际上，如果插件没有特定的生命周期管理需要，它就根本不需要实现此接口。

在每个插件中都存在一个由框架管理的 OSGi Bundle。Bundle 是 OSGi 模块单元。从根本上来说，Bundle 只是安装在平台中的文件（资源和代码）集合。每个 Bundle 都具有它自己的

Java 类装入器，并且包括用于启动、停止和卸载它本身的协议。从 Eclipse 平台的角度来说，Bundle只不过是一个实现类。插件开发者不扩展 Bundle 类，但是使用Plugin或其他 BundleActivator 实现来表示插件。



在 Eclipse 3.0 之前，提供了插件注册表（IPluginRegistry）来提供类似的信息。例如，可以查询它以获得系统中所有插件的插件描述符。现在，不推荐使用此注册表，应该使用 BundleContext 来达到此目的。平台注册表现在专用于提供有关扩展和扩展点的信息。

1.6 开源的选择

Eclipse 是一个开源的、可扩展的集成开发环境，已经吸引了业界的很多注意力，Eclipse 的支持者源源不断。Eclipse 有着非常强大的功能。对于编码、调试、重构、单元测试等，Eclipse 可以与任何一种 IDE 匹敌，甚至比它们还要好。与免费的 JBuilder X Foundation 版本类似，Eclipse 也有代码补足、代码模板的功能，以及对重构、Ant、CVS 和 JUnit 的支持。实际上，用户可能会发现 Eclipse 对很多特性的支持都比收费的 IDE 更好。例如，Eclipse 具有更多的重构功能，能够更好地支持将本地代码与 CVS 服务器进行合并。

除了这些基本功能之外，Eclipse 的优点之一就是它的源代码是开放的，可扩展性很好，可以扩充很多插件，开源插件和商业插件都可以用来扩充 Eclipse。如果用户需要基本的 Eclipse IDE 所不具备的特性，一般都可以找到需要的插件。在大部分情况下，用户都可以找到免费的插件。

除了免费的插件之外，用户还可以找到很多用于 Eclipse 的价格适中的商业插件。如果只需要一部分高级特性，却无法找到满意的免费插件，就可以从第三方供应商那里购买这些插件，而不用去购买完整的企业 IDE 包，这样会更经济。

数据表明 Eclipse 是现在最流行的 Java 开发环境之一，EclipseSource 进行了相关调查，统计出人们使用 Eclipse 的原因。调查结果有如下几个方面。

（1）价格低廉的解决方案：Eclipse 是一个免费的软件，任何人都可以在www.eclipse.org 网站免费获取，并且没有任何 license 的限制。

（2）开源平台：Eclipse 是一个开源的软件，任何人都可以免费下载 Eclipse 源代码，并且可以在源代码的基础上进行修改。

（3）有许多种类的插件供使用：Eclipse 已经推出很长时间了，基于 Eclipse 的插件已经相当多，且功能非常强大，选择余地也比较大。

（4）可扩展性：Eclipse 是一个可扩展的平台。任何人都可以利用 Eclipse 的插件扩展机制扩展别人的扩展点，并且提供自己的扩展点，供别人扩展使用。

（5）可以跨平台运行：因为 Java 是跨平台运行的，而 Eclipse 是用 Java 开发的 IDE 环境，所以 Eclipse 是可以跨平台运行的，而且在 Eclipse 中开发出来的产品都是可以跨平台的。

（6）不会被某个特定的供应商锁定：Eclipse 联盟是一个非营利性质的组织，现在大多数主流的软件开发商都已经加入了 Eclipse 联盟，并为 Eclipse 的发展做贡献。

（7）容易学习和使用：Eclipse 是免费开源的平台，任何人都可以在此基础上扩展。Eclipse

会不断吸收其他软件的优点，并集成到平台中，使之不断完善。

由于 Eclipse 的流行性和开源特性，它已经在 IDE 领域成为一把尖刀，一方面，Eclipse 在一步一步完善自己的功能，另一方面，Eclipse 正在大刀阔斧抢占 IDE 开发的应用领域。

1.7 小　　结

Eclipse 平台是 IBM 向开发源码社区捐赠的开发框架，它之所以出名并不是因为 IBM 宣称投入开发的资金总数为 4 000 万美元，而是因为如此巨大的投入所带来的成果：一个成熟的、精心设计的、可扩展的体系结构。Eclipse 的价值是它为创建可扩展的集成开发环境提供了一个开放源码平台。这个平台允许任何人构建与环境和其他工具无缝集成的工具。工具与 Eclipse 无缝集成的关键是插件。除了小型的运行时内核之外，Eclipse 中的所有东西都是插件。

本章介绍了 Eclipse 中的基本概念，这些都是 Eclipse 比较基础的知识。如果读者不太明白也没关系，但希望能牢记于心，在 Eclipse 的使用开发过程中，将会慢慢理解这些概念。

第 2 章　Eclipse 安装及 Java 开发

在国内，大多数用户很乐于将 Eclipse 当做 Java IDE 来使用，本章将介绍如何安装 Eclipse，以及如何用 Eclipse 进行 Java 开发。

本章从如下几个方面进行介绍。

- Eclipse 开发环境的安装
- Eclipse 中的窗口及菜单
- 用 Eclipse 开发 Java 程序
- Eclipse 快捷键

2.1　Eclipse 开发环境的安装

要开发 Java 程序，必须要有 JDK（Java Development Kit）的支持，在安装 Eclipse 之前用户应该先安装 JDK，下面将介绍如何安装 JDK 和 Eclipse。

2.1.1　JDK 安装

JDK 是整个 Java 的核心，包括了 Java 运行环境（Java Runtime Envirnment），一堆 Java 工具和 Java 基础的类库（rt.jar）。不论什么 Java 应用服务器实质都是内置了某个版本的 JDK。因此掌握 JDK 是学好 Java 的第一步。最主流的 JDK 是 Sun 公司发布的 JDK，除了 Sun 之外，还有很多公司和组织都开发了自己的 JDK，例如 IBM 公司开发的 JDK，BEA 公司的 Jrocket，还有 GNU 组织开发的 JDK 等等。下面将介绍 Sun 公司 JDK 的安装使用。

（1）下载 JDK

JDK 又叫做 J2SE（Java 2 SDK Standard Edition），可以从 Sun 的 Java 网站（http://java.sun.com/j2se/downloads.html）上下载，建议下载最新版本。本书代码可以在 JDK 1.4 以上的版本运行。

（2）安装 JDK

下载后的 JDK 是一个可执行安装程序，安装 JDK 步骤非常简单，执行安装程序后按步骤安装即可。

（3）配置 Java 的路径

默认安装完毕后会在 C:\Program Files\Java\目录下安装一套 JRE，笔者把 JDK 安装在 C:\j2sdk1.4.2 下面（其中也会包括一套 JRE）。然后需要在环境变量 Path 的最前面增加 Java 的路径 C:\j2sdk1.4.2\bin。这样 JDK 就安装好了，且 Eclipse 可以通过 Path 的 JDK 路径找到 JDK。



Eclipse 会通过 Path 找到 JDK，如果用户安装了 Oracle 等其他工具，这些工具也会带 JDK，可能会和安装的 Eclipse 的 JDK 有冲突。如果出现这个问题，只要在 Path 中把刚安装的路径置于其他设置的路径之前即可。

2.1.2 Eclipse 安装

安装 Eclipse 非常简单，用户只要到 Eclipse 的官方网站下载 Eclipse 的 SDK，下载以后解压到自己想要的目录中，例如 D:\Eclipse，直接运行 eclipse.exe，第一次运行的时候会自动寻找 JDK 并完成相应的配置，然后就进入了 Eclipse 的主界面。

Eclipse 解压以后就可以运行，没有 Windows 程序中烦琐的安装过程，另外 Eclipse 也不会在系统中注册，用户的系统中可以同时存在多个 Eclipse 的版本，而且不会相互冲突。Eclipse 的安装是非常“干净”的（不会在系统以及注册表中写入安装信息），如果想删除 Eclipse，只需要删除 Eclipse 文件即可。

本书使用的 Eclipse 版本是 3.1.1，本书的源程序在 Eclipse 3.0 以上的版本都可以运行。读者也可以从 http://www.eclipse.org 下载“eclipse-SDK-3.1.1-win32.zip”文件，解压以后即可运行 Eclipse。

2.2 Java 开发工具 JDT

Eclipse 的 SDK 中自带了 Java 开发工具，它也是以插件的形式提供的。用户运行 Eclipse 后，就可以通过 Eclipse 提供的 Java 开发工具进行 Java 开发。为了更好地开发 Java 项目，Java 开发工具提供了一系列的视角、视图、编辑器和其他插件，本节将简要介绍用户在开发中常用到的视角、视图和编辑器。

2.2.1 视角

为了更好地组织 Java 程序的开发和调试工作，Java 开发工具把 Java 的视图分成几种类型进行组织，用户可以通过不同的视角（有的书上也称为透视图）打开和视角相关联的视图，从而更好地组织程序的开发工作，Java 的视角有如下几类。

（1）Java 视角

设计成用于处理 Java 项目的视角。它包含一个编辑器区域和下列视图。

- 包资源管理器
- 层次结构
- 大纲
- 搜索
- 控制台
- 任务

（2）Java Broswing 视角

设计成用于浏览 Java 项目结构的视角。它包含一个编辑器区域和下列视图。

- 项目
- 包
- 类型
- 成员

（3）Java Type Hierarchy 视角

设计成用于浏览类型层次结构的视角。可对类型、编译单元、包、项目或源文件夹打开该视角，它由“层次结构”视图和编辑器组成。

（4）Debug 视角

设计成用于调试 Java 程序的视角。它包括一个编辑器区域和下列视图。

- 调试
- 断点
- 表达式
- 变量
- 显示
- 大纲
- 控制台

2.2.2 视图

为了方便开发 Java 程序，Java 开发工具为工作台提供了一些视图，用户通过这些视图能够快速完成 Java 开发的工作，Java 开发工具中有如下视图。

（1）Package Explorer 视图

Package Explorer 视图显示工作台中 Java 项目的 Java 元素层次结构。它提供“导航器”中显示资源的相应视图。元素层次结构是从项目的构建类路径中派生出来的。对于每个项目，其源文件夹和引用的库都显示在树中。可以打开和浏览内部和外部 JAR 文件的内容。

（2）Hierarchy 视图

Hierarchy 视图允许用户查看类型的完整层次结构、只查看它的子类型或只查看它的超类型。

（3）Projects 视图

Projects 视图显示 Java 项目、源文件夹、外部和内部库。在此视图中存在的源文件夹和库（包括内部的和外部的）是不可扩展的。当选择它们时，它们的内容就会显示在 Packages 视图中。

（4）Packages 视图

Packages 视图显示当前选择的 Java 项目、源文件夹或库中的 Java 包列表。通常，使用 Projects 视图来进行此选择。

（5）Types 视图

Types 视图显示当前选择包中的 Java 类型列表。通常，使用 Packages 视图来进行此选择。

（6）Members 视图

Members 视图显示类型、编译单元或 CLASS 文件的内容。通常，使用“Types”视图来进行此选择。

2.2.3 编辑器

Java 编辑器提供了用于编辑 Java 代码的特殊功能部件。与编辑器相关联的是特定于 Java 的 Outline 视图，它显示活动 Java 编译单元的结构。它是在用户编辑编译单元时更新的。

可以对二进制数 CLASS 文件打开 Java 编辑器。如果 JAR 文件包含具有源代码连接的 CLASS 文件，则编辑器会显示相应的源文件。

编辑器包含下列功能部件：

- 语法突出显示
- 内容 / 代码辅助
- 代码格式化
- 导入辅助
- 快速修正
- 集成的调试功能部件

调用 Java 编辑器的最常用方法是使用弹出菜单从“导航器”或“包”资源管理器中打开 Java 文件或单击文件（单击或双击，这取决于用户首选项）。还可以通过从其他视图打开 Java 元素（如类型、方法或字段）来打开编辑器。

2.3 在 Eclipse 中开发 Java 程序

Eclipse 通过 JDT 来开发 Java 程序，JDT 只是 Eclipse 的一组插件。Eclipse 中也有项目和文件等资源的概念，JDT 中的项目和文件是通过扩展 Eclipse 中的功能实现的，要建立 Java 程序首先要建立 Java 的项目（包含有 Java 的特性），在 Java 项目的基础上再添加 JDT 的其他资源。

2.3.1 Java 项目

Java 项目包含用于构建 Java 程序的源代码和相关文件。它有一个相关联的 Java 构建器，在更改 Java 源文件时，Java 构建器可以对这些 Java 源文件进行增量编译。

Java 项目还维护其内容的模型。此模型包含关于 Java 元素的类型层次结构、引用和声明信息。当用户更改 Java 源代码时，此信息会不断地更新。对内部 Java 项目模型的更新与 Java 构建器无关；尤其是在执行代码修改时，即使关闭了自动构建，模型也仍将反映最新的项目内容。

添加Java 项目步骤如下。

（1）选择“New”→“Project”菜单，打开新建项目对话框，如图 2-1 所示。

（2）选择 Java Project 选项，单击“Next”按钮，打开新建 Java 项目对话框，如图 2-2 所示。

在新建 Java 项目对话框输入项目的名字，可以单击“Next”按钮打开项目配置对话框，也可以单击“Finish”按钮按默认值建立一个 Java 项目。另外可以选择把源文件和编译后的

Class 文件分开，如图 2-2 的 Project layout 所示，这样 Eclipse 默认会创建一个 src 的目录，在 src 目录下用户可以添加自己的 Java 程序。



图 2-1　新建项目对话框



图 2-2　新建 Java 项目对话框

2.3.2　新建“Hello，world”程序

用户可以在 Java 项目中添加自己的 Java 程序，添加“Hello，world”程序的步骤如下。

（1）建立一个包（package），用包来组织 Java 的源程序（如果不新建包，也可以用源文件的根目录作为类的包，即默认包），如图 2-3 所示。

（2）在包创建 Java 类文件，类名为 HelloWorld，如图 2-4 所示。

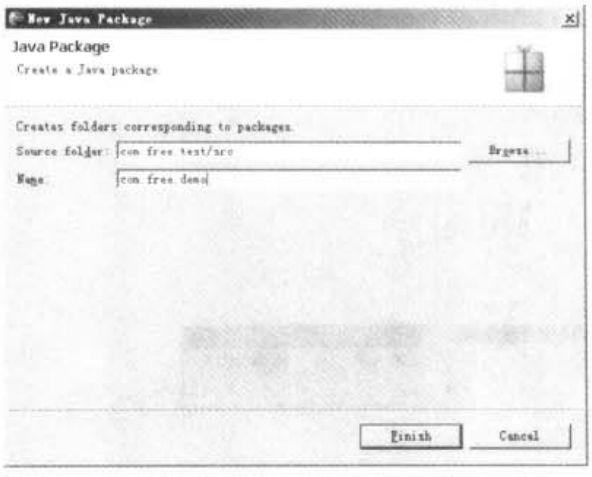


图 2-3　创建包对话框



图 2-4　创建 Java 类

创建 Java 的包名可以用“.”来表示目录结构，如“com.free.demo”表示创建“com\free\demo”

形式的目录结构。另外，创建 Java 类时可以指定是否创建 main 函数、父类及类的一些其他属性。

上面的步骤已经在“com.free.demo”包下面创建了一个带 main 函数的 HelloWorld 类，现在可以在 main 函数中添加 Java 代码实现“Hello，world”的例子了，代码如例程 2-1 所示。

例程 2-1　HelloWorld.java

```
package com.free.demo;

public class HelloWorld {

    public static void main(String[] args) {

        String text = "Hello world ";
        for (int i = 0; i < 10; i++){
            System.out.println(text + i);
        }
    }
}
```

如上例所示，“package com.free.demo”表示程序所在的包名，另外有一个静态的 main 函数，这就是一个简单的可运行程序。

2.3.3　运行程序与调试

在创建了简单的程序后，接下来就是让程序运行起来，在 HelloWorld 的编辑区域单击鼠标右键，选择“Run As”→“Java Application”命令就可以运行“Hello，World”程序，如图 2-5 所示。

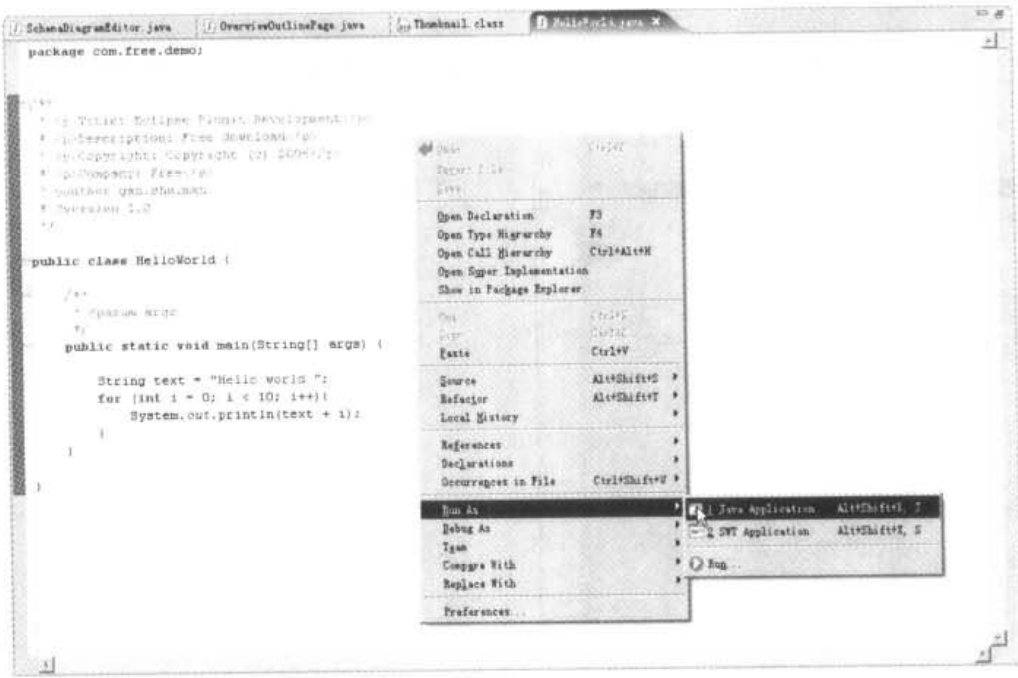


图 2-5　运行 Java 程序

此程序的运行结果会在控制台打印出 “Hello world ” 信息，但如果想跟踪程序的运行，就需要调试的功能了，可以通过如下步骤调试程序。

（1）设置断点：在想设断点的代码行按【Ctrl+Shift+B】快捷键可以在此行设置一个断点，当程序运行到此行时会暂停（也可双击行的左边框设置断点）。

（2）运行调试程序：在 HelloWorld 的编辑区域单击鼠标右键，选择“Debug As”→“Java Application”命令就可以调试“Hello，World”程序。

（3）查看变量的值：当程序运行到断点处时，用户可以选择某一个变量，单击鼠标右键选择“Watch”菜单，则在 Expressions 视图中可以看到变量的值，也可以选择“Inspect”菜单，直接查看变量的值，如图 2-6 所示，变量 *i* 的值等于 3。

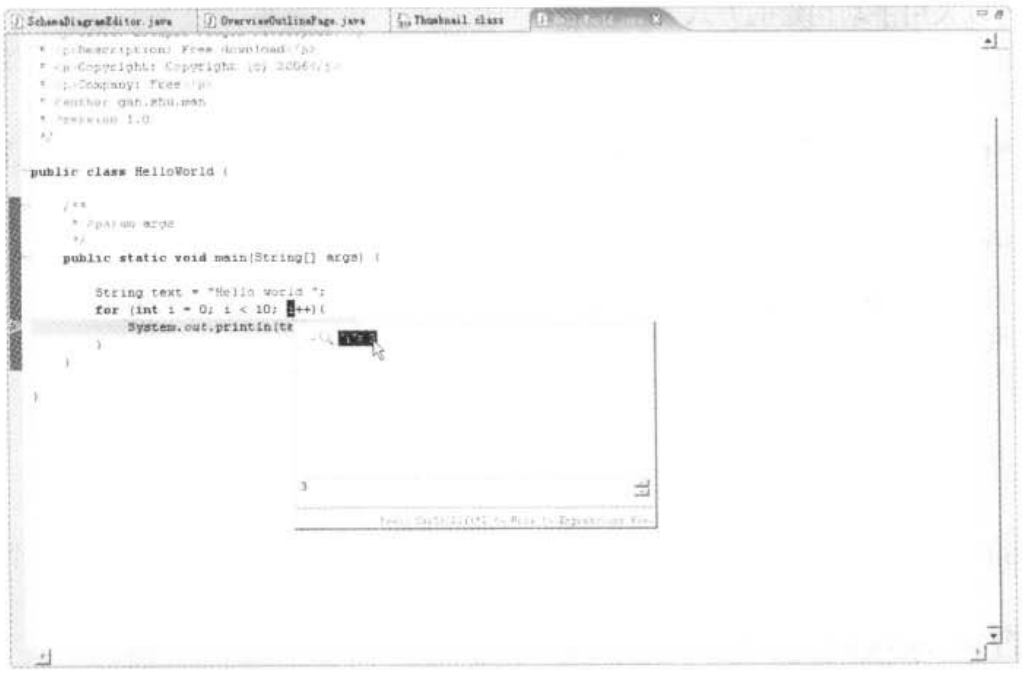


图 2-6　调试 Java 程序

2.3.4　类的构建路径

Java 程序可以引用源代码文件、其他 Java 项目和 JAR 文件，可以使用文件系统路径或通过使用引用网络位置的变量来指定 JAR 文件。Java 程序通过类的构建路径来查找引用的 Java 代码。

类的构建路径是用来查找源代码所引用的类的路径。在编译期间，使用此路径来搜索项目外部的类。为每个项目指定构建类路径。在项目属性中，将其称为“Java 构建路径”。

用户可设置项目的构建路径，步骤如下。

（1）选择项目，并从它的上下文菜单中选择“Properties”菜单。

（2）在 Properties 对话框中，选择“Java Build Path”。

（3）选择 Projects 选项页，设置项目依赖的其他项目。

（4）选择 Libraries 选项页，设置构建路径上的 Jar 包和 Class 类。

（5）选择 Order and Export 选项页，设置类的查找顺序。

Java 项目可以引用其他的 Java 项目，可以在 Projects 选项页设置，另外可以通过 Libraries

选项页添加第三方的 Jar 包到当前项目的构建路径中。

> **注意** 如果在构建路径的同一位置有两个相同的类，例如，两个不同的 Jar 包的同一路径上有相同的类，则构建路径按“Order and Export”中指定的顺序查找。

2.3.5 项目构建

当用户修改了 Java 源代码以后，如果 Eclipse 的设置为自动构建，且当 Java 元素（文件或其他资源）改变后，Eclipse 会自动编译修改的 Java 元素。如果没有设置 Eclipse 自动构建选项，用户可以采用手动构建的方式。

用户可以通过 Eclipse 的“Project”菜单的子菜单设置构建的方式，Java 项目有如下几种构建方式。

（1）“Build Automatically”菜单：选定时表示自动构建项目，一般来说可以设置为自动构建。

（2）“Build All”菜单：增量构建修改后的所有项目。

（3）“Build Project”菜单：增量构建当前项目。

（4）“Clean”菜单：清除所有的项目，重新构建。

2.4 其 他 功 能

JDT 功能非常强大，本书无法对此一一介绍，下面只介绍在开发中常用到的几个功能。

2.4.1 查看源代码

如果想查看某一个类的实现，用户可以按住【Ctrl】键不放（或按【F3】键），再单击方法名、类名或属性名就能查看源代码的实现，如图 2-7 所示。



图 2-7 查看源代码

但是有些代码引用的是 Jar 中的类或方法，例如 JDK 的 Integer 类，跟踪到 Integer 类后，只能看到 Integer 编译的 Class 文件，如图 2-8 所示。

如果想打开 Class 文件对应的源文件，还需要指定源文件的路径，单击上图的“Attach Source”按钮可以指定要查看的源文件，如图 2-9 所示。



图 2-8　打开 Class 文件

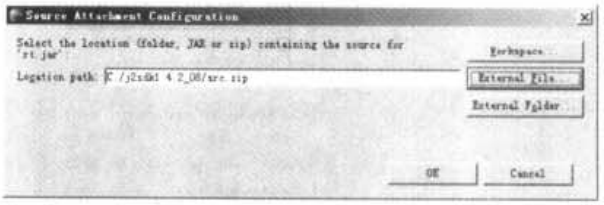


图 2-9　设置源代码路径

2.4.2　重构

如果要对 Java 元素进行重命名，应该怎样做呢？例如，用户写了一个类 Demo，而且在项目中很多地方引用到了这个类，如果把类名从 Demo 改成 Test 的话，其他引用到 Demo 的地方都要修改，假设你的项目很大的话，这种重命名的工作将会浪费用户很多的时间，而且可能还会出错。

在项目中需要重命名包名、文件名和方法名是比较常见的事，Eclipse 的重构功能为用户很好地解决了重命名的问题，当用户重命名 Java 元素后，可以选择性地更新和 Java 元素相关联的引用，如图 2-10 所示。

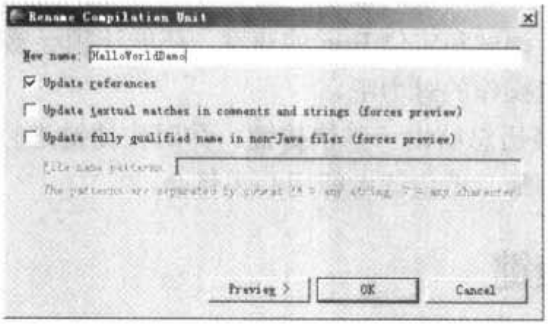


图 2-10　重命名 Java 文件

图 2-10 选择了更新 Java 文件中的所有引用，还可以选择更新注释和字符串中的文本，以及更新非 Java 文件中配置的字符串（例如插件的 Plugin.xml 文件）。

2.4.3　生成 Javadoc 文档

Java 的 Javadoc 可以帮助用户生成帮助文档，在 Eclipse 中也提供了利用 Javadoc 生成帮助

文档的机制，选择菜单“Project”→“Generate Javadoc”，打开生成 Java 文档的对话框，可以在生成 Java 文档对话框中设置 javadoc.exe 的位置、生成 Java 文档的项目、生成 Java 成员的类型和输出文档的位置等信息生成 Java 文档，如图 2-11 所示。

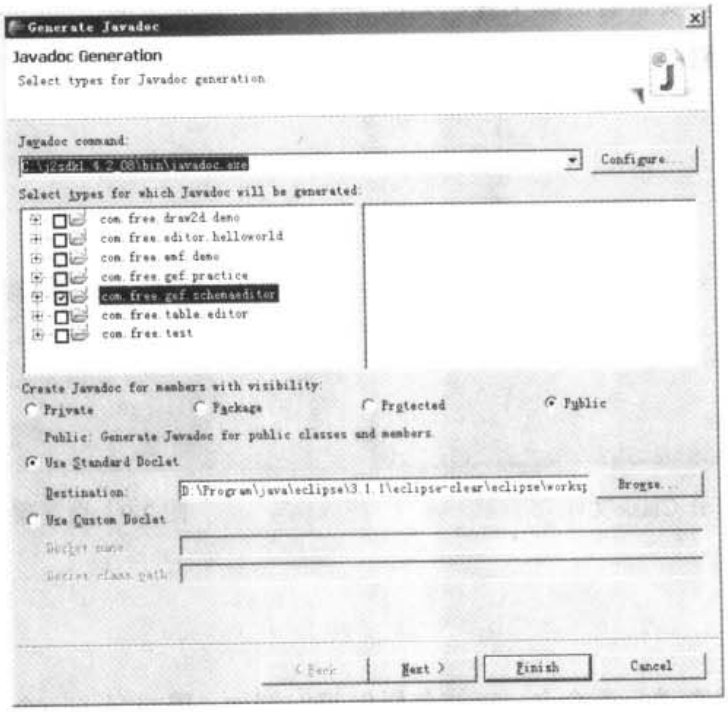


图 2-11　生成 Java 文档

2.5　Eclipse 快捷键

想成为编程“高手”吗？如果说有捷径的话，不妨多花点时间在本节的快捷键使用上。作为一个使用 Eclipse 进行 Java 开发的编程“高手”，一些常用的快捷键是必须掌握的，它不但能够大大加快程序编写、调试和定位 Bug 的速度，还能让领导感觉你的“打字速度”，何乐而不为呢？尝试一下全键盘操作的魅力吧。

本节将从笔者的亲身经历总结出一些快捷键，并对快捷键进行分类，而且按笔者的喜好进行排列，并加上相关的解释，希望读者能用心去体会。

2.5.1　编辑相关快捷键

Eclipse 的编辑功能非常强大，掌握了 Eclipse 快捷键功能，能够大大提高开发效率。Eclipse 中有如下一些和编辑相关的快捷键。

（1）【ALT+/】

此快捷键为用户编辑的好帮手，能为用户提供内容的辅助，不要为记不全方法和属性犯愁，当记不全类、方法和属性的名字时，多体验一下【ALT+/】快捷键带来的好处吧。

（2）【Ctrl+O】

显示类中方法和属性的大纲，能快速定位类的方法和属性，在查找 Bug 时非常有用。

（3）【Ctrl+/】

快速添加注释，能为光标所在行或所选定行快速添加注释或取消注释，在调试的时候可能总会需要注释一些东西或取消注释，现在好了，不需要每行进行重复的注释。

（4）【Ctrl+M】

窗口最大化和还原，用户在窗口中进行操作时，总会觉得当前窗口小（尤其在编写代码时），现在好了，试试【Ctrl+M】快捷键。

（5）【Ctrl+D】

删除当前行，这也是笔者的最爱之一，不用为删除一行而按那么多次的删除键。

（6）【Ctrl+1】

快速修正，有了此快捷键，就不用写“try”、“catch”等冗长、重复的代码了，选定代码试试【Ctrl+1】快捷键。

2.5.2 查看和定位快捷键

在程序中，迅速定位代码的位置，快速找到 Bug 的所在，是非常不容易的事，Eclipse 提供了强大的查找功能，可以利用如下的快捷键帮助完成查找定位的工作。

（1）【Ctrl+K】、【Ctrl++Shift+K】

快速向下和向上查找选定的内容，从此不再需要用鼠标单击查找对话框了。

（2）【Ctrl+Shift+T】

查找构建路径中的可找到类，不要为找不到类而痛苦，而且可以使用“*”、“？”等通配符。

（3）【Ctrl+Shift+R】

查找构建路径中的所有文件，也可以使用通配符。

（4）【Ctrl+Shift+G】

查找类、方法和属性的引用。这是一个非常实用的快捷键，例如要修改调用某个方法位置的代码，可以通过【Ctrl+Shift+G】快捷键迅速定位所有的调用位置。

（5）【Ctrl+Shift+O】

快速生成 import，当从网上拷入一段程序后，不知道如何 import 进所调用的类，试试【Ctrl+Shift+O】快捷键，一定会有惊喜。

（6）【Ctrl+Shift+F】

格式化代码，书写格式规范的代码是每一个程序员的必修之课，当看见某段代码极不顺眼时，选定后按【Ctrl+Shift+F】快捷键可以格式化这段代码，如果不选定代码则默认格式化当前文件。

（7）【ALT+Shift+W】

查找当前文件所在项目中的路径，可以快速定位浏览器视图的位置，如果想查找某个文件所在的包时，此快捷键非常有用（特别在比较大的项目中）。

（8）【Ctrl+L】

定位到当前编辑器的某一行。

（9）【Alt+←】、【Alt+→】

后退历史记录和前进历史记录，在跟踪代码时非常有用，用户可能查找了几个有关联的

地方，但可能记不清楚了，可以通过这两个快捷键定位查找的顺序。

（10）【F3】

快速定位光标位置的某个类、方法和属性。

（11）【F4】

显示类的继承关系。

2.5.3 调试快捷键

Eclipse 中有如下一些和运行调试相关的快捷键。

（1）【Ctrl+Shift+B】：在当前行设置断点或取消设置的断点。

（2）【F11】：调试最后一次执行的程序。

（3）【Ctrl+F11】：运行最后一次执行的程序。

（4）【F5】：跟踪到方法中，当程序执行到某方法时，可以按【F5】键跟踪到方法中。

（5）【F6】：单步执行程序。

（6）【F7】：执行完方法，并执行到方法的后一条语句。

（7）【F8】：继续执行，到下一个断点或程序结束。

2.5.4 常用编辑器快捷键

通常文本编辑器都提供了一些和编辑相关的快捷键，在 Eclipse 中也可以通过这些快捷键进行文本编辑。

（1）【Ctrl+C】：复制。

（2）【Ctrl+X】：剪切。

（3）【Ctrl+V】：粘贴。

（4）【Ctrl+S】：保存文件。

（5）【Ctrl+Z】：撤销。

（6）【Ctrl+Y】：重复。

（7）【Ctrl+F】：查找。

2.5.5 其他快捷键

Eclipse 中还有很多快捷键，无法一一列举，用户可以通过帮助文档找到它们的使用方式，另外还有几个常用的快捷键如下。

（1）【Ctrl+F6】：切换到下一个编辑器。

（2）【Ctrl+Shift+F6】：切换到上一个编辑器。

（3）【Ctrl+F7】：切换到下一个视图。

（4）【Ctrl+Shift+F7】：切换到上一个视图。

（5）【Ctrl+F8】：切换到下一个视角。

（6）【Ctrl+Shift+F8】：切换到上一个视角。

2.6 小 结

本章介绍了 Eclipse 的安装、JDT 的功能和在 Eclipse 中开发、调试 Java 程序，读者应该掌握一个 Java 项目的内容及如何配置项目的类路径。

另外，快捷键也是本章的重点，虽然说快捷键只是一个辅助的功能，但如果掌握好则能让用户有“如虎添翼”的感觉。快捷键也不是一时半会都能掌握的，要在项目中反复试验，要做到心中有“键”。

第 3 章　Eclipse 的使用

Eclipse 除了很小的内核之外，其他的都是插件，要想更好地利用 Eclipse 进行开发，掌握插件的使用是有必要的，本章将介绍 Eclipse 中插件的安装，以及通过开发一个简单的 Web 应用让读者了解 Eclipse 和插件的强大之处。

本章包括如下内容。

- Eclipse 插件的安装
- 开发简单的 Web 应用

3.1　Eclipse 插件安装

Eclipse 最有魅力的地方就是它的插件体系结构，由于有了插件，Eclipse 系统的核心部分在启动的时候要完成的工作十分简单：启动平台的基础部分和查找系统的插件。

初学者可能会对插件有恐惧心理，认为这是 Eclipse 底层的东西。其实不然，安装插件非常容易，安装插件时注意如下的几个问题就可以了。

（1）插件的依赖关系

如果用户要安装的插件还需要其他插件的支持，则需要安装依赖的插件才能使新安装的插件正常运行。例如用户要安装一个图形编辑的插件，但此插件需要图形编辑框架（GEF）插件的支持，只有先安装了 GEF 才能使新的插件安装成功。

（2）缓存

Eclipse 会通过缓存的机制加载插件，有时用户刚安装的插件可能没有正常启动（特别是 links 方式安装），清除缓存后再重新启动 Eclipse 就行了。清除缓存最简单的方式是删除 Eclipse 的 configuration 目录下的所有文件夹（保留 config.ini 文件）。

（3）版本

当用户安装的插件要求某一个特定版本的 Eclipse，或依赖某个特定版本的插件时，也可能会使插件安装不成功。

所以，用户在安装插件之前要看看插件依赖的插件和特定版本信息的帮助文档。本节将介绍安装插件的几种方式，以及如何查看插件是否安装成功。

3.1.1　拷贝安装

Eclipse 是通过一种非常“干净”的安装方式进行安装的，不会因为安装 Eclipse 影响其他软件的运行，安装 Eclipse 的插件也是如此。

拷贝安装是一种最简单的安装插件方式，用户只要把插件的压缩文件解压以后拷贝到 Eclipse 的安装目录即可。例如，“GEF-ALL-3.1.zip”是 GEF 插件的压缩包，其中包含了 Eclipse 目录，Eclipse 目录下面包含了 plugins 目录和 features 目录等，如图 3-1 所示。



图 3-1 GEF 插件压缩包结构

用户可以解压以后把 plugins 目录和 features 目录拷贝到安装 Eclipse 的根目录即可，即用 plugins 目录和 features 目录覆盖 Eclipse 根目录下的相应子目录（不会删除以前的文件，只会把新添加的文件拷贝过去）。

3.1.2 links 安装

通过 links 方式安装插件是一种既“干净”又有效的安装方式，也是笔者极力推荐的一种安装方式。试想一下，通过拷贝的安装方式安装插件，如果同一个 Eclipse 下安装的插件太多，想删除某几个插件怎么办？用户就得在 Eclipse 的 plugins 目录中找到所有插件目录，把它们删除掉，另外，还得删除 feature 下的所有相关安装选项，这是多么痛苦的事情呀。

links 安装方式为用户解除了这种烦恼，通过 links 安装方式，用户不需要把所有的插件都拷贝在 Eclipse 目录中，只需要编写一个文本文件，通过文件指定插件的位置，让 Eclipse 找到所指的插件就可以了。当用户要删除插件时，只需要删除文本文件或文本文件中的引用即可，通过 links 安装插件的步骤如下。

（1）在 Eclipse 目录（ECLIPSE_HOME）下创建一个 links 目录。

（2）在你的%ECLIPSE_HOME%\links 目录下创建一个 link 文件（只要是文本文件就可以）。

（3）把 link 文件指向插件所在的目录，例如“path=D:/Program/java/eclipse/Plugin-download/GEF-ALL-3.1”，路径的分隔符为“/”或“\\”，另外，一个文件中可以指向几个插件，多写几行“path=...”即可，也可以在 links 目录下面多写几个 link 文件，笔者是通过把每一个插件写到一个文件的方式安装插件的，这样删除插件时可直观地删除文件，如图 3-2 所示。

（4）插件所在目录结构和插件在 Eclipse 中的目录结构一样，例如 GEF 插件解压以后的结构为“GEF-ALL-3.1/eclipse/...”，link 文件指向“GEF-ALL-3.1”所在目录即可，Eclipse 启动以后会找到“GEF-ALL-3.1”目录下 Eclipse 子目录中所有的插件。

用户如果要删除相关插件，只要删除 link 文件或删除文件中的引用即可。

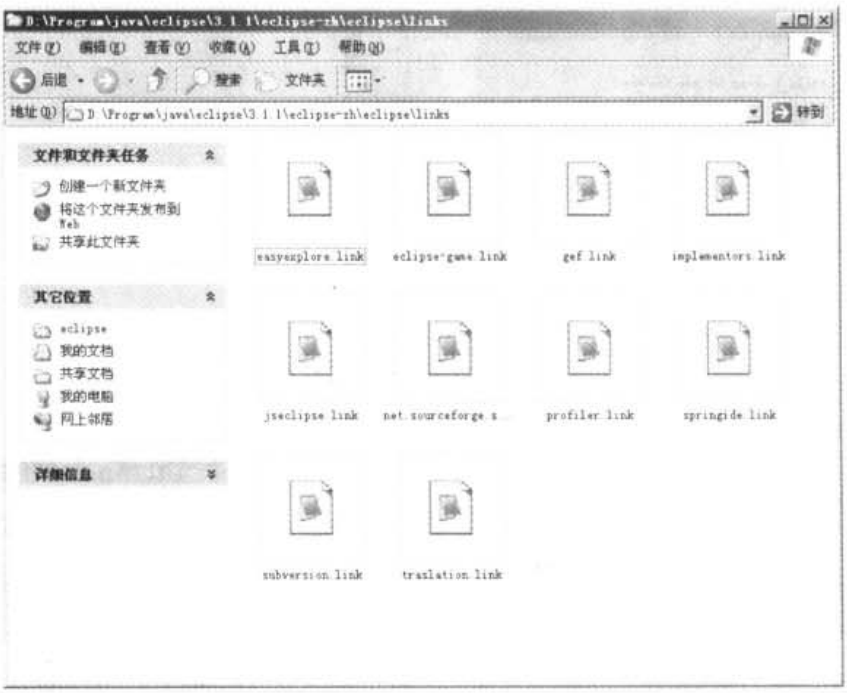


图 3-2　links 方式安装插件

注意　如果删除 link 文件或 links 目录后，重新启动 Eclipse 失败，可以清除缓存（删除 configuration 目录中的子目录即可）。

3.1.3　update 安装

除了拷贝安装方式和 links 安装方式外，Eclipse 还能在联网的情况下通过连接插件服务器，直接下载插件并安装插件。这种方式操作简单，用户只要知道插件的更新地址就可以。通过解析更新地址的 XML 文件，Eclipse 知道要到什么位置找到插件更新包。另外，Eclipse 还能判断当前安装插件的版本是否正确。

通过 update 方式安装插件非常简单，具体步骤如下。

（1）选择“Help”→“Software Updates”→“Find and Install”菜单，弹出对话框如图 3-3 所示。

（2）选择第二项，单击“Next”按钮，打开显示 Update 站点的对话框，如图 3-4 所示。

（3）在显示站点对话框单击“New Remote Site”按钮，打开指定 Update 站点的对话框，如图 3-5 所示。

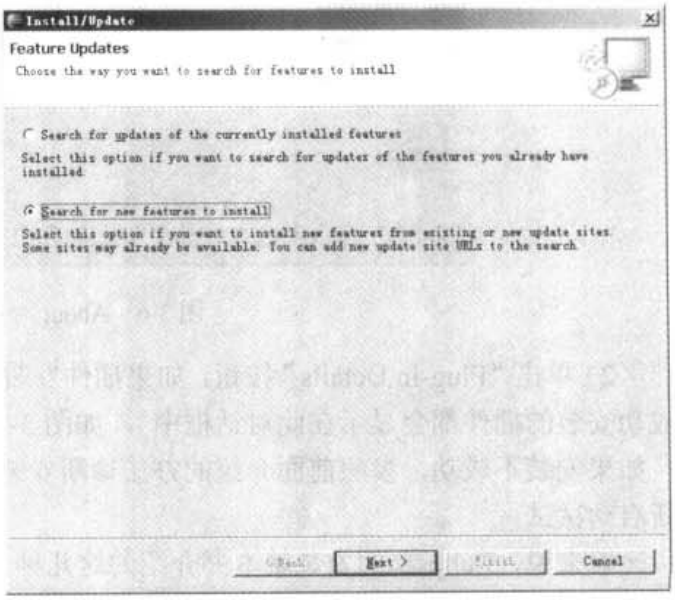


图 3-3　查找新的 Update 站点

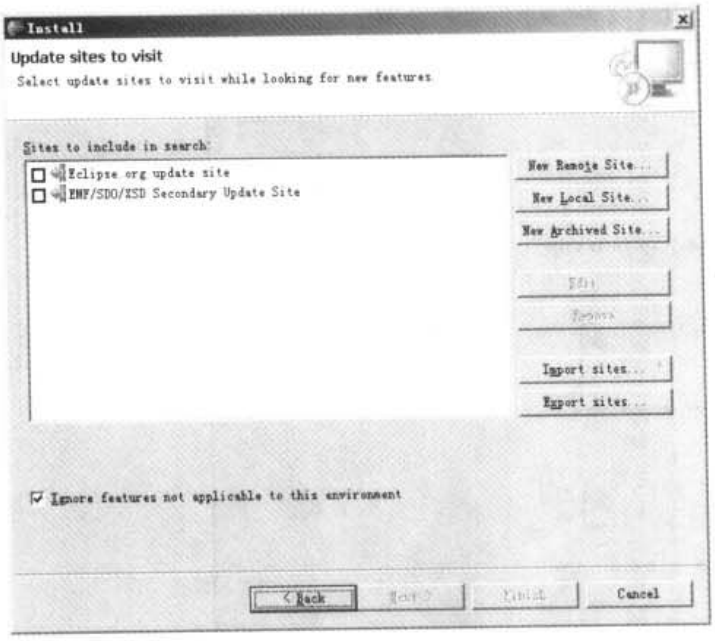


图 3-4　显示 Update 站点

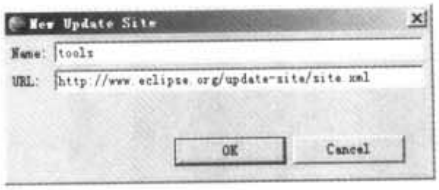


图 3-5　添加新的 Update 站点

（4）添加新的站点后，按 Eclipse 的更新向导就可以完成插件安装了。

3.1.4　查看插件

用户安装了插件后，如果想确认插件是否安装成功，可以重新启动 Eclipse，查看插件步骤如下。

（1）选择“Help”→“About Eclipse SDK”菜单，弹出的对话框如图 3-6 所示。

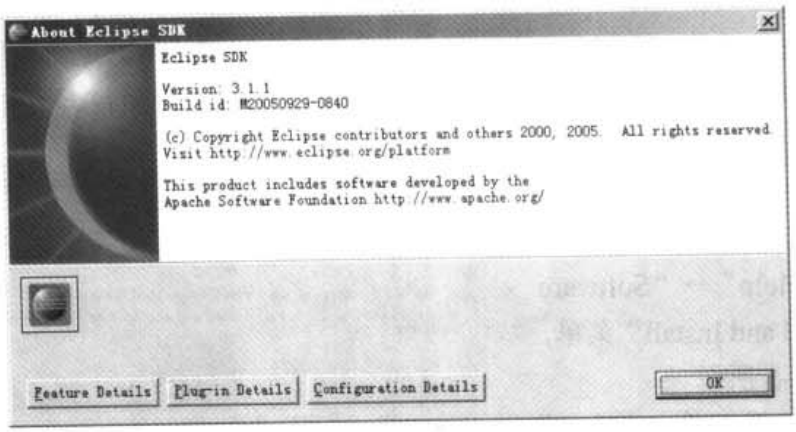


图 3-6　About 对话框

（2）单击“Plug-in Details”按钮，如果插件安装成功，将会显示在插件显示对话框中（所有成功安装的插件都会显示在此对话框中），如图 3-7 所示。

如果安装不成功，参照前面介绍的办法诊断安装的步骤是否有误，或清空 Eclipse 的缓存重新启动试试。

一般来说，Eclipse 的安装就本书介绍的这几种方式，但有些商业插件可能会通过安装程序安装插件，例如下一节开发 Web 应用的 MyEclipse 插件就是通过安装程序进行安装。

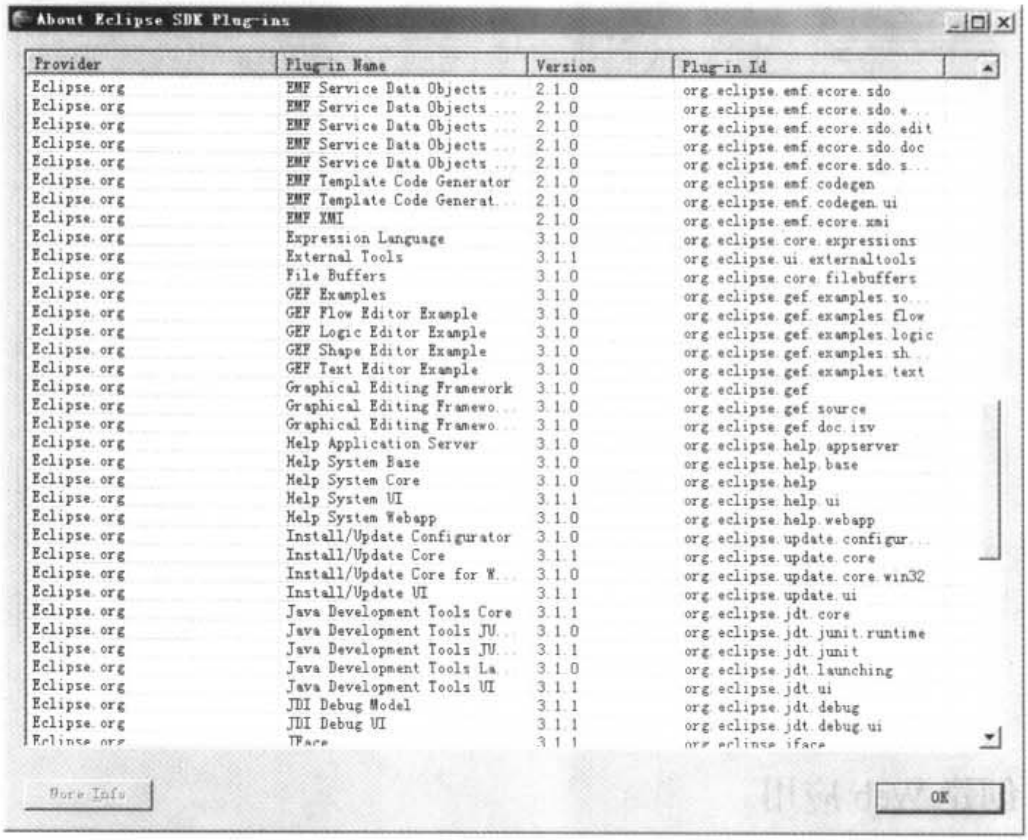


图 3-7 插件列表对话框

3.2 开发简单的 Web 应用

很多的 Java 程序员都做过 Web 方面的开发，Eclipse 中 Web 开发的插件有很多，本节将通过使用 MyEclipse 来开发一个简单的 Web 应用，让读者了解通过 Eclipse 以及 Eclipse 插件给用户开发带来的便利。

3.2.1 安装配置 MyEclipse

MyEclipse 是一个商业的插件，可以通过运行它的安装程序安装 MyEclipse。另外安装好 MyEclipse 后，要先购买注册码注册后才能使用。

如果要开发 Web 应用，用户需要先安装好 Web 服务器，本节演示中使用的 Web 服务器为 Tomcat 5.0。安装好 Tomcat 以后，要配置好 MyEclipse 中服务器的位置，只有配置好了 Web 服务器，MyEclipse 才知道把创建的 Web 应用部署到什么位置。配置 Web 服务器很简单，可以选择菜单“Window”→“Preferences”打开首选页对话框，并配置 Tomcat 服务器的位置，如图 3-8 所示。

当配置好 Tomcat 服务器后，用户可以把自己新建的 Web 项目部署在 Tomcat 中，并在 Eclipse 中运行调试 Web 程序。

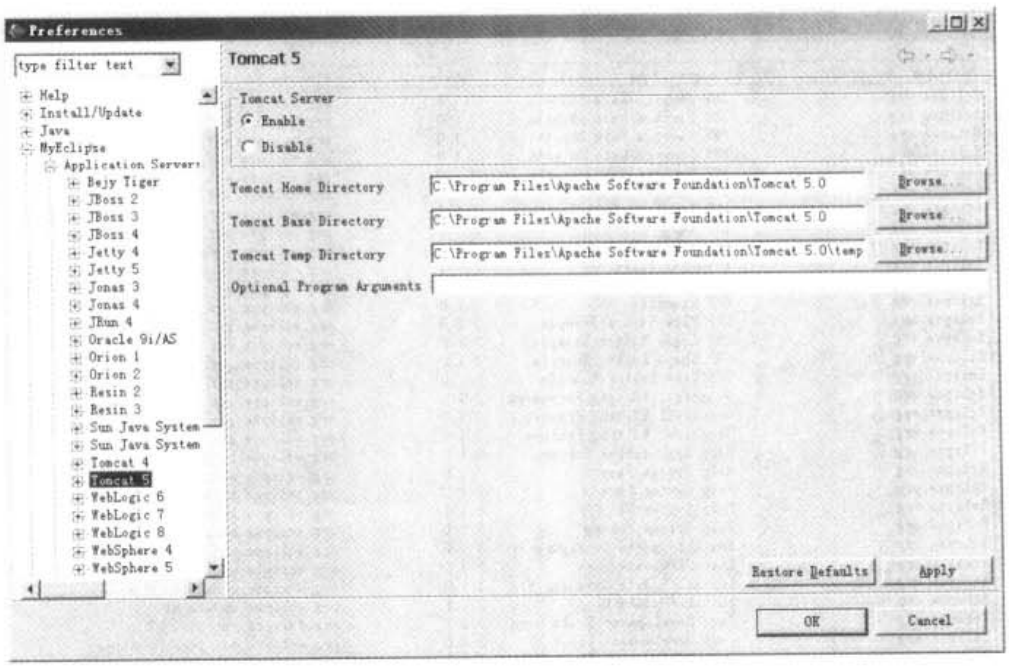


图 3-8　配置 Web 服务器

3.2.2　创建 Web 应用

要在 Eclipse 中开发 Web 应用，首先要在 Eclipse 中建立一个 Web 项目，在 Web 项目的基础上建立并部署自己的应用。创建 Web 应用的步骤如下。

（1）创建 Web 项目。选择新建项目的向导，创建 Web 应用，如图 3-9 所示。

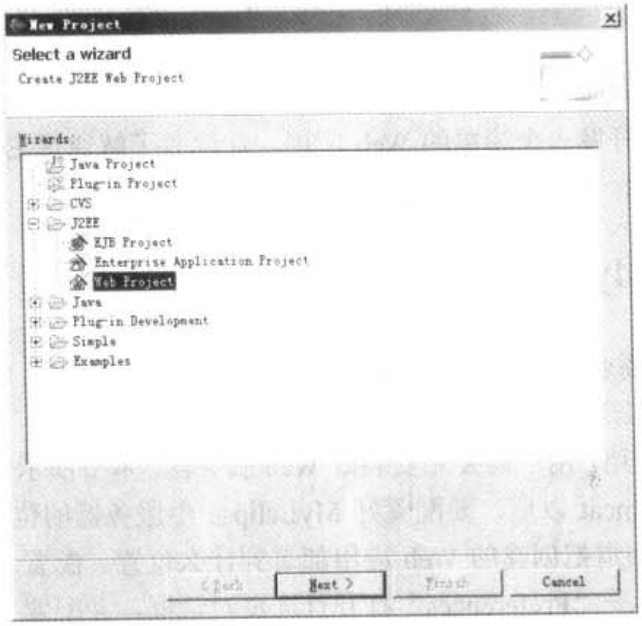


图 3-9　创建 Web 应用

（2）在创建的 Web 项目中添加演示的 Servlet 响应类和 HTML 文件。

本节通过一个文件上传的示例演示如何开发一个 Web 应用，示例中包含一个上传文件的 HTML 文件、一个响应上传文件的 Servlet 响应类。另外还要通过 Web.xml 文件配置请求的映

射关系。

首先，创建一个 HTML 文件，通过网页上传文件，代码如例程 3-1 所示。

例程 3-1 upload.html

```
<html>
<head>
<title>文件上传</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body>
<b>文件上传</b></font></p>

<form name="UploadForm" enctype="multipart/form-data"
   method="post" action="fileupload">
   <input type="file" name="File1" size="20" maxlength="20"> <br>
   <input type="file" name="File2" size="20" maxlength="20"> <br>
   <input type="submit"value="上传">
</form>

</body>
</html>
```

upload.html 文件通过 post 请求，把要上传的文件发送到路径为 fileupload 的服务器端程序。

其次，创建服务器端处理文件上传请求的 Servlet，并把上传的文件保存到指定的文件中，服务器端代码如例程 3-2 所示。

例程 3-2 upload.java

```
package com.free.test;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.commons.fileupload.*;

public class Upload extends HttpServlet {

   private String uploadPath = "C:\\upload\\";       // 用于存放上传文件的目录
   private String tempPath = "C:\\upload\\tmp\\";    // 用于存放临时文件的目录

   public void doPost(HttpServletRequest request, HttpServletResponse respons
e)
      throws IOException, ServletException
      {
       try {

          DiskFileUpload fu = new DiskFileUpload();
          // 设置最大文件大小，这里是 90MB
```

```
            fu.setSizeMax(91943040);
            // 设置缓冲区大小，这里是4KB
            fu.setSizeThreshold(4096);
            // 设置临时目录
            fu.setRepositoryPath(tempPath);

            // 得到所有的文件
            List fileItems = fu.parseRequest(request);
            Iterator i = fileItems.iterator();
            // 依次处理每一个文件
            while(i.hasNext()) {
                FileItem fi = (FileItem)i.next();
                // 获得文件名，这个文件名包括路径
                String fileName = fi.getName();
                if(fileName!=null) {
                    // 在这里可以记录用户和文件信息
                    fi.write(new File(uploadPath+"/"+fi.getFieldName()));
                        System.out.println(fi.getFieldName());
                }
            }
            // 跳转到上传成功提示页面
        }
        catch(Exception e) {
            // 可以跳转出错页面
            e.printStackTrace();
        }
    }
}
```

Upload 类继承于 HttpServlet 类，处理 HTTP 请求。另外，Upload 类利用 Apache 下面的 FileUpload 组件实现了文件的上传，用户只要把 FileUpload 所用到的 Jar 包拷贝到 Web 项目 Web-INF 目录下的 lib 目录，MyEclipse 就会处理类的查找路径。

最后，要把 upload.html 中 action 对应的路径映射到 Upload 类，这一步要配置 web.xml 文件，添加路径到 Servlet 类的映射关系，代码如例程 3-3 所示。

例程 3-3　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
      <servlet-name>Upload</servlet-name>
      <servlet-class>com.free.test.Upload</servlet-class>
    </servlet>

    <servlet-mapping>
```

```
    <servlet-name>Upload</servlet-name>
    <url-pattern>/fileupload</url-pattern>
  </servlet-mapping>

</web-app>
```

通过 web.xml 文件，添加了响应路径到 Upload 类的映射关系，用户可以通过浏览器地址“http://localhost:8080/upload/upload.html”调用上传文件的网页，并能处理文件的上传。

3.2.3　部署并运行 Web 应用

通过上面的步骤，现在已经建立了一个完整的处理文件上传的 Web 应用，接下来要做的就是把 Web 应用部署到 Tomcat 中，让用户可以通过创建的 Web 应用上传文件。

在 MyEclipse 中部署 Web 应用非常简单，步骤如下。

（1）在浏览器视图中用鼠标右键单击想部署的项目，选择“MyEclipse”→“Add and Remove Project Deployments”菜单，打开部署项目的对话框。

（2）单击“Add”按钮，添加用户想部署的服务器，在此选择配置的 Tomcat 服务器，如图 3-10 所示。

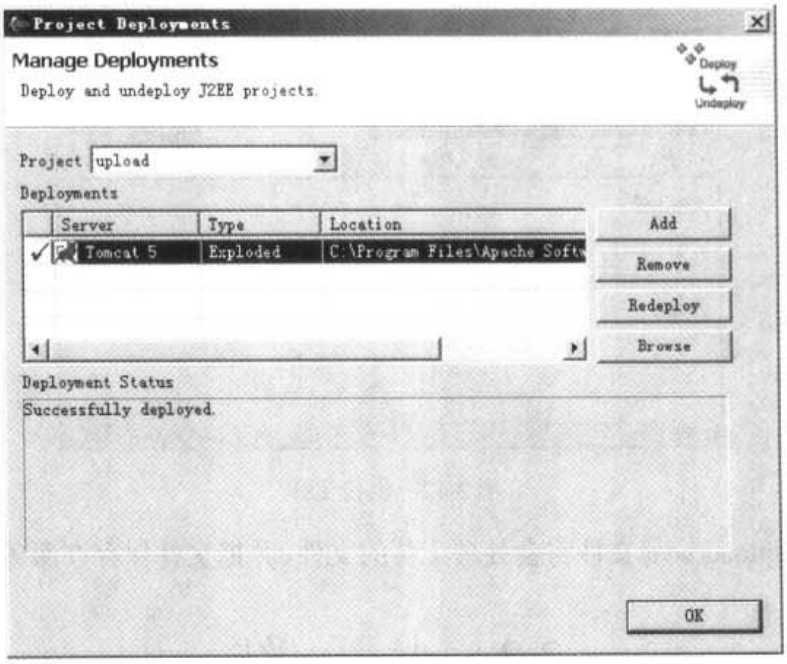


图 3-10　部署 Web 项目

（3）单击“OK”按钮，Web 项目就部署成功了。

在部署成功文件上传的 Web 应用以后，MyEclipse 会把 Web 项目中的文件拷贝到 Tomcat 的 webapps 下的相应目录，现在可以通过 Eclipse 启动 Tomcat 服务器（用户也可以不在 Eclipse 中启动 Tomcat，但通过 Eclipse 启动 Tomcat 的话，用户可以在 Eclipse 调试 Servlet 程序，这是非常有用的），也可以通过工具栏的下拉菜单按钮启动 Tomcat 服务器，如图 3-11 所示。

启动 Tomcat 后，用户可以通过网页 upload.html 文件上传文件了，在浏览器中输入如下地址“http://localhost:8080/upload/upload.html”，打开网页，选择上传的文件，如图 3-12 所示。

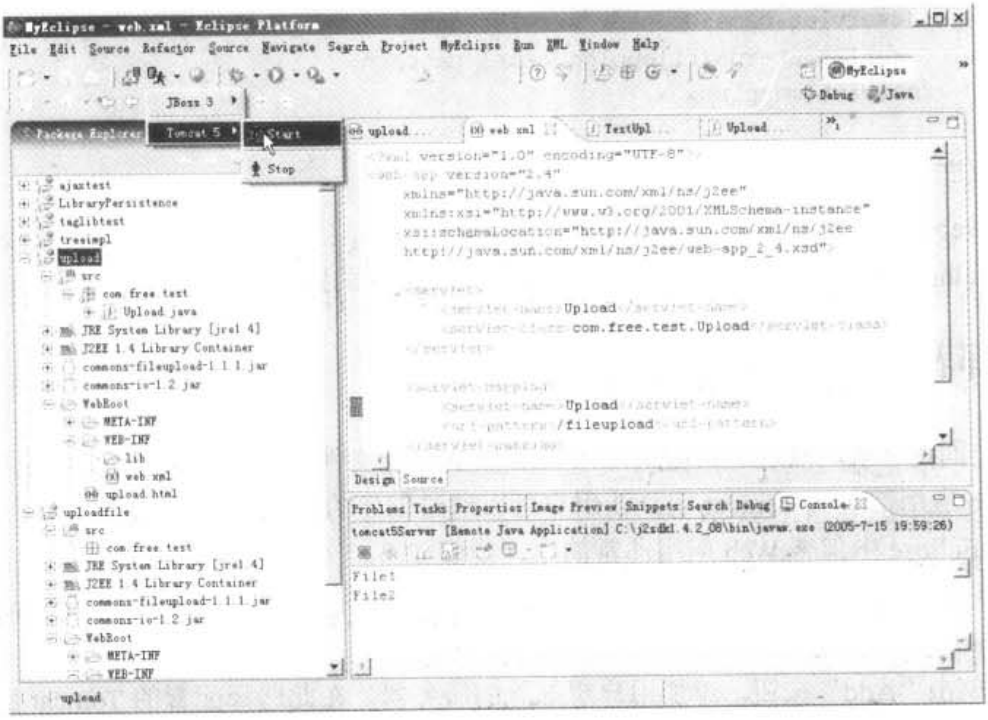


图 3-11　启动 Tomcat

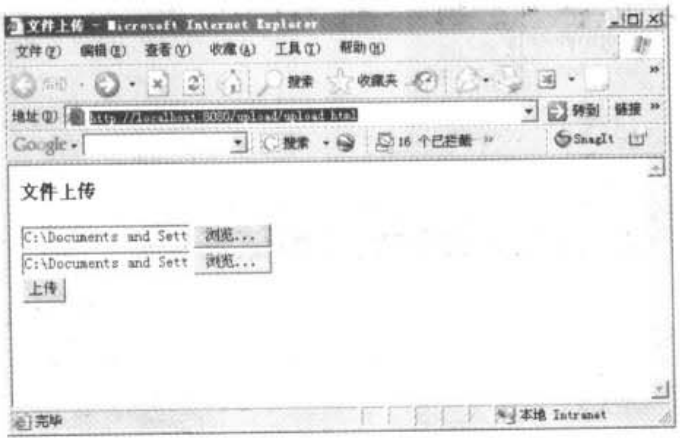


图 3-12　上传文件

选择上传后，upload.java 文件将会处理上传的文件，并把文件保存在服务器上的相应目录。

3.3　小　　结

本章介绍了 Eclipse 插件的安装，利用 Eclipse 插件开发相关的 Web 应用。在本章，读者要求掌握插件的安装方式，很多用户都因为安装插件出问题而放弃了一些插件的使用，通过本章的学习，相信会解除读者安装插件的烦恼。

开发 Web 应用只是学习使用 Eclipse 开发的一个入门，希望读者在今后的使用和开发过程中能够不断探索，继续提高。

Eclipse 插件开发方法与实战

第 2 部分　SWT/JFace 开发篇

用 Java 做过开发的程序员对 Swing 肯定不陌生，Swing 提供了基于 Java 的一套 GUI 的库。但 Java 语言的声望和它在桌面应用程序（GUI 程序）所取得的成就显然极不相符，至今仍然很少能看到非常成功的 Java 桌面程序。而 SWT 无论是在性能和外观上，都超越了 Sun 公司提供的 AWT 和 Swing。

本部分介绍了 SWT/JFace 开发，其中包括 SWT/JFace 的组件、事件和布局的实现。SWT/JFace 是 Eclispe 的基础，Eclipse 的 Workbench 就是建立在 SWT/JFace 之上的。另外，JFace 是在 SWT 之上开发的，它和 SWT 形成一个交集，其中 SWT 提供最原始的组件。只有掌握了 SWT/JFace 的开发才能更好地进行插件的开发，就如同盖房子需要砖头一样重要，如果说插件是房子，SWT 就是砖头。

第 4 章　SWT/JFace 入门

SWT（Standard Widget Toolkit）本身仅仅是 Eclipse 组织为了开发 Eclipse IDE 环境所编写的一组底层图形界面 API。或许是无心插柳，或许是有意为之，至今为止，SWT 无论在性能上还是外观上，都超越了 Sun 公司提供的 AWT 和 Swing。

本章主要从以下几个部分进行介绍。

- SWT/JFace 简介
- SWT 的“Hello world”程序
- JFace 的“Hello world”程序
- SWT 中使用 Swing 组件

4.1　SWT/JFace 简介

Eclipse 是一种通用工具平台。它是一个开放的、可用于任何东西的可扩展 IDE，它为工具开发人员提供了灵活性及对软件技术的控制。Eclipse 为开发人员提供了生产大量 GUI 驱动的工具和应用程序的基础。而这项功能的基础就是 GUI 库 SWT 和 JFace。

4.1.1　SWT/JFace 概述

SWT（Standard Widget Toolkit）是 Eclipse 中的窗口小部件工具箱，它提供可移植的 API，并与底层本机操作系统图形用户界面平台紧密集成。另外，SWT 定义在所有受支持平台上提供的公共可移植 API，并尽可能地使用本机窗口小部件在每个平台上实现该 API，这允许工具箱在所有平台上维护一致的编程模型时立即反映底层操作系统图形用户界面外观中的任何更改。

JFace 用来在 SWT 库顶部提供常见的应用程序用户界面功能。JFace 并不试图“隐藏”SWT 或者替换它的功能。它提供一些类和接口，以处理与使用 SWT 来对动态用户界面编程相关联的许多常见任务。

SWT/JFace 是 Eclispe 的基础，Eclipse 的 Workbench 就是建立在 SWT/JFace 之上的。另外，JFace 是在 SWT 之上开发的，它和 SWT 形成一个交集，其中 SWT 提供最原始的组件，如图 4-1 所示。

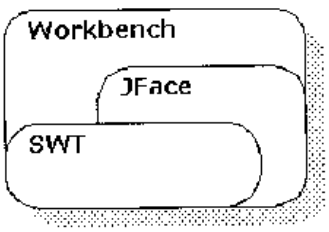


图 4-1　SWT/JFace 和 Workbench 的关系

JFace 对 SWT 进行了扩展，把用户熟悉的一些组件进行了封装，在开发中用户可以尽可能地用 JFace 组件来开发自己的应用。JFace 程序和 SWT 程序类似，只不过 JFace 把常用的功能进行了提炼，使用户不必太关心 SWT 的一些细节。

4.2.2 SWT/JFace 基本特性

SWT 是一个库，它创建了 Java 版的本地主机操作系统 GUI 控件。它依赖于本机实现，这意味着基于 SWT 的应用程序具有以下几个关键特性。

（1）它们的外观、行为和执行类似于“本机”应用程序。

（2）所提供的窗口小部件（Widget）反映了主机操作系统上提供的窗口小部件（组件和控件）。

（3）主机 GUI 库的任何特殊行为都在 SWT GUI 中得到反映。

这些目标使得 SWT 不同于 Java 技术的 Swing，Swing 的设计目标是消除操作系统的差异。SWT 库反映了主机操作系统的基本窗口小部件。JFace 库有助于向 SWT 应用程序中添加大量服务。JFace 并没有隐藏 SWT，它只是扩展了 SWT。SWT 最重要的扩展之一是，将应用程序的数据模型与显示及更改它的 GUI 隔离开来。

SWT 中有如下一些基本的组件。

（1）Widget：基本的 SWT GUI 组件（类似于 Java AWT 中的 Component 和 Swing 中的 JComponent），Widget 是一个抽象类。

（2）Control：拥有操作系统的对等物的窗口小部件，Control 是一个抽象类。

（3）Composite：包含其他控件的控件（类似于 Java AWT 中的 Container 和 Swing 中的 JPanel）。

（4）Item：其他控件包含的窗口小部件（该控件可能不是复合控件），如列表和表。Item 是一个抽象类。

这些窗口组件（或小部件）被安排在继承层次结构中。其中 Widget 是底层的类，继承关系如图 4-1 所示。

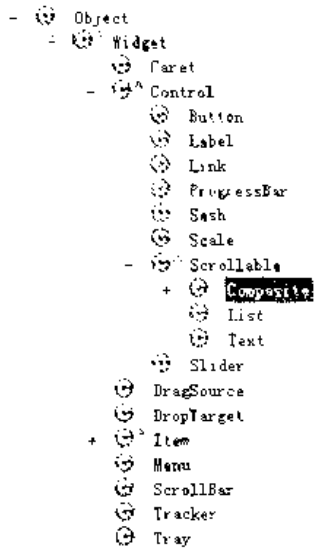


图 4-2　Widget 类的继承关系

几乎所有 SWT GUI 都是从某些基础部分开始创建的。所有 SWT 窗口组件都可以在 org.eclipse.swt.widget 或 org.eclipse.swt.custom 包中找到（一些 Eclipse 插件还在其他包中提供了定制的窗口组件）。窗口组件包中包含一些基于操作系统控件的控件，而定制包中则包含一些超出操作系统控件集之外的控件。一些定制的软件包控件类似于窗口小部件包中的控件。为了避免命名冲突，定制控件的名称都是以“C”开始的（例如，比较 CLabel 与 Label）。

在 SWT 中，所有控件（除了一些高级控件，比如 shell，将在后面进行讨论）在创建的时候都必须有一个父控件（一个复合实例）。在创建的时候，这些控件被自动“添加”到父控件中，这与必须明确添加到父控件中的 AWT/Swing 中的控件有所不同，自动添加产生了一种“自上而下”地构造 GUI 的方法。这样，所有控件都可以采用一个复合父控件（或者一个子类）作为构造函数的参数。

大多数控件都有一些必须在创建时设置的标记选项。因此，大多数控件还有另外一个构造函数参数，通常称为样式或风格。该参数提供了设置这些选项的标记。所有这些参数值都是 static final int，并且都是在 org.eclipse.swt 包的 SWT 类中定义的。如果不需要任何参数，则可以使用 SWT.NONE 值。

4.2 SWT“Hello world”程序

SWT 是一套独立的库，用户可以在 SWT 的基础上很容易地建立自己的应用，本节的目的就是要在 SWT 上建立一个简单的“Hello world”程序，读者可以在这个简单的程序上建立自己的应用。

4.2.1 加入 SWT 依赖的包

SWT 为用户提供了一套 API，用户如果要使用 SWT 进行开发，必须要把所需的 Jar 包放在 Eclipse 项目的 ClassPath 中。在此笔者采用新建一个插件项目，插件项目会自动引用 SWT 相关的包，步骤如下。

（1）运行 Eclipse。

单击“File”→“New”→“Other”菜单项，在弹出的对话框中选择 Select 对话框左边的 Plug-in Development 向导。如图 4-3 所示，选择 Plug-in Project。

（2）单击“Next”按钮，弹出新建对话框，输入项目名称。此处使用了“com.free.swtjface.demo”。单击“Next”按钮后弹出新建对话框页，如图 4-4 所示。

（3）单击“Next”按钮，在接下来的对话框中单击“Finish”按钮，建立一个空的插件项目在此项目中，Eclipse 中已经自动引入了此插件所需要依赖的包，其中包括 SWT/JFace 所用到的包如图 4-5 所示。

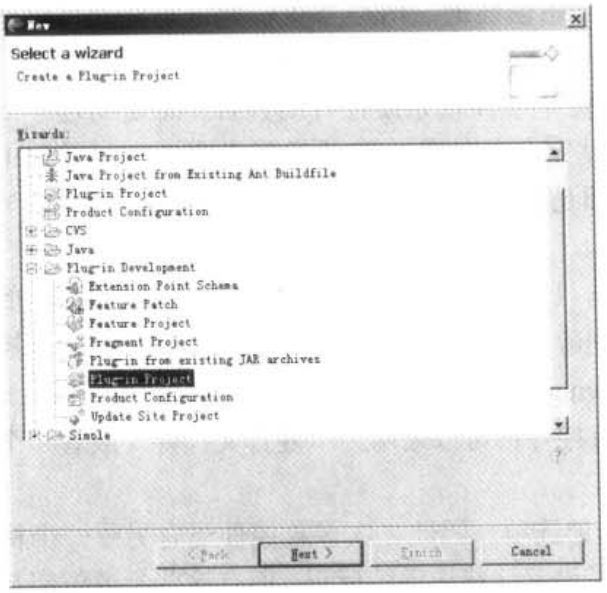


图 4-3　新建插件对话框

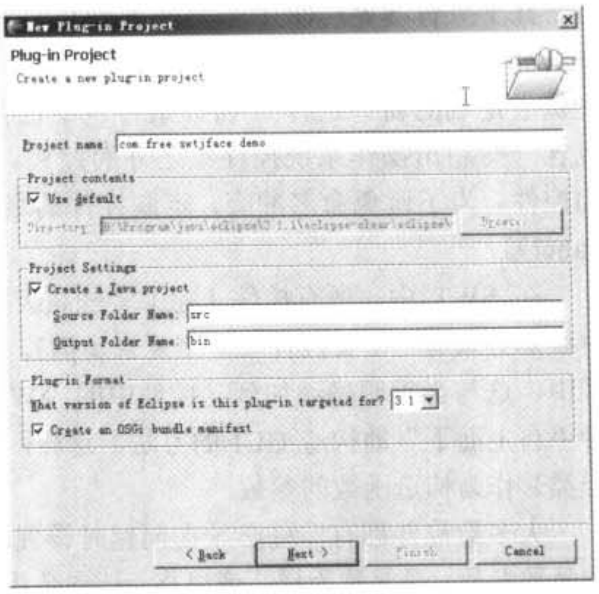


图 4-4　新建项目

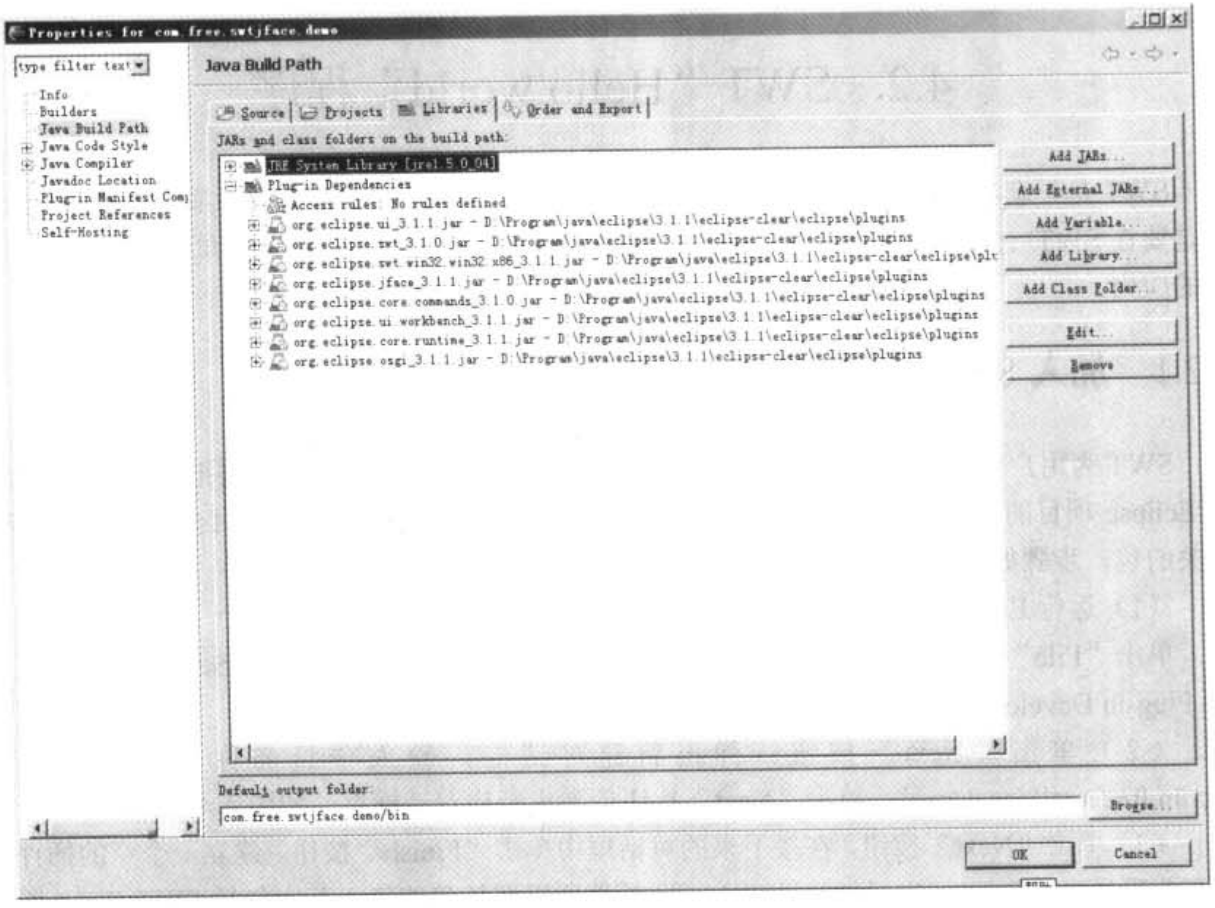


图 4-5　插件依赖的包

如果用户想加入 SWT/JFace 相关的包，也可以手动加入下面几个包“org.eclipse.swt.win32.win32.x86_3.1.1.jar”、“org.eclipse.swt_3.1.0.jar”和“org.eclipse.jface_3.1.1.jar”到插件的类路径中。

4.2.2 “Hello world”程序

现在SWT所依赖的包已经加入到了项目的ClassPath中，接下来写一个SWT程序，代码如例程4-1所示。

例程4-1 HelloWorldSwt.java

```
package com.free.helloworld;

import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class HelloWorldSwt {

    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("hello world! Window");
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        display.dispose();
    }

}
```

这段程序虽然很简单，但是它反映了我们书写SWT程序的步骤，如下。

（1）创建一个Display对象。Display对象是SWT和操作系统通信的桥梁。

（2）创建一个或者多个Shell对象。可以认为Shell代表了程序的窗口。

（3）在Shell内创建各种部件（widget），如按钮、下拉框和列表等。

（4）对各个部件进行初始化（外观、状态等），同时为各种部件的事件创建监听器（listener）。监听器可以监听窗口的消息，此处没有添加监听器，将会在后面的章节重点介绍监听器的注册。

（5）调用Shell对象的open()方法以显示窗体。

（6）对各种事件进行监听并处理，直到程序发出退出消息。

（7）调用Display对象的dispose()方法以结束程序。

4.2.3 运行“Hello world”程序

SWT程序的运行要通过JNI调用相应的操作系统控件，运行SWT程序和运行Java应用程序有点不同，在Eclipse 3.1中用户可以选择运行SWT程序即可，步骤如下。

（1）打开Java的默认视图，在“Hello world”程序的文件上单击鼠标右键。

（2）选择“SWT Application”菜单，如图4-6所示。

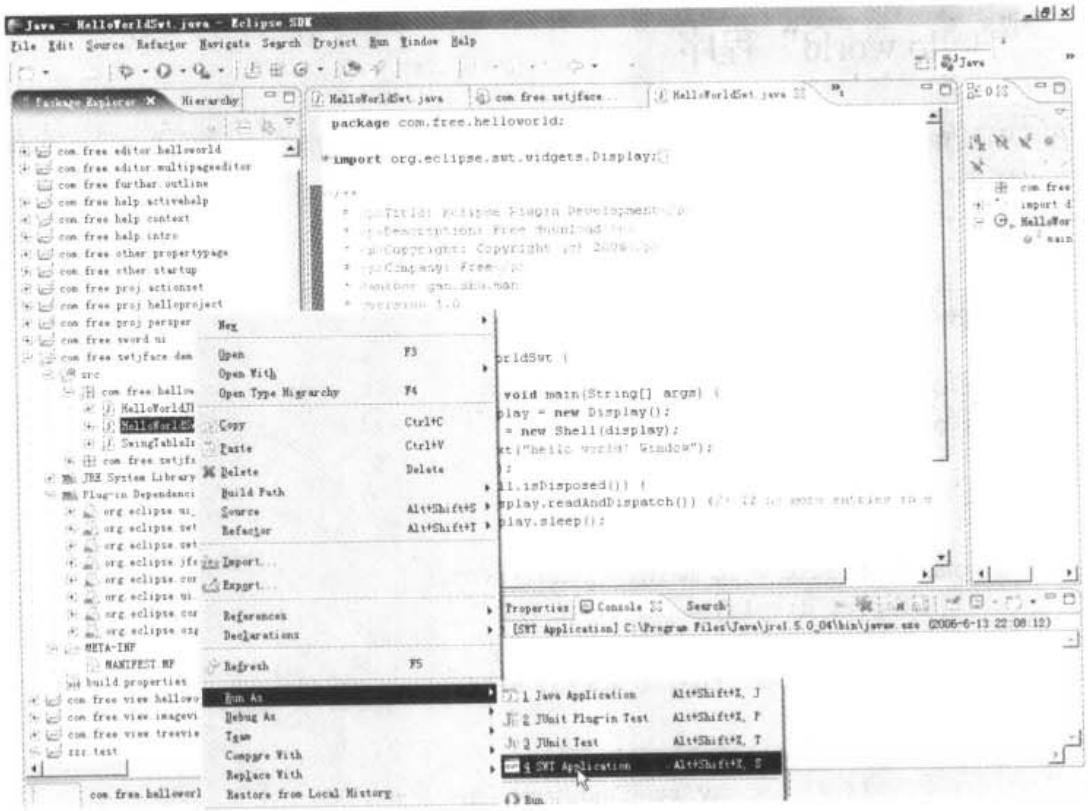


图 4-6　运行 SWT 程序

（3）单击“SWT Application”菜单运行，运行效果如图 4-7 所示。



图 4-7　SWT“Hello world”程序

4.3　JFace“Hello world”程序

JFace 是一个用户界面工具箱，它提供很难实现的、用于开发用户界面功能部件的 helper 类，JFace 在原始的窗口小部件系统的级别之上运行。JFace 使用户可以专注于实现特定插件的功能，而不必花费精力来处理底层窗口小部件系统或者解决几乎在任何用户界面应用程序中都很常见的问题。

4.3.1　“Hello world”程序

JFace 的应用程序相对来说更简单，它通过 ApplicationWindow 类实现应用，ApplicationWindow

把和操作系统交互的细节封装起来，用户只需要关心自己窗口的建立，程序代码如例程 4-2 所示。

例程 4-2 HelloWorldJFace.java

```
package com.free.helloworld;

import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;

public class HelloWorldJFace extends ApplicationWindow {
public HelloWorldJFace() {
        super(null);
    }

    protected Control createContents(Composite parent) {
        getShell().setText("hello world! Window");
        parent.setSize(400, 250);
        return parent;
    }

    public static void main(String[] args) {
        HelloWorldJFace helloWorldApp = new HelloWorldJFace();
        helloWorldApp.setBlockOnOpen(true);
        helloWorldApp.open();
        Display.getCurrent().dispose();
    }
}
```

其中，JFace 的应用程序可以通过重载 createContents(Composite parent)方法添加窗口的组件到 parent 组件中。

4.3.2 运行“Hello world”程序

运行 JFace 的程序和运行 SWT 程序一样，程序运行效果如图 4-8 所示。

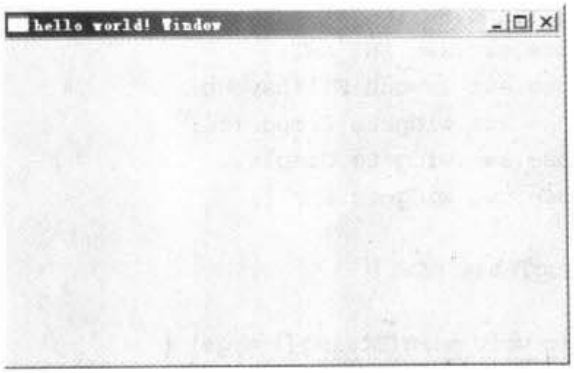


图 4-8 JFace“Hello world”程序

4.4 在 SWT 中使用 Swing 组件

SWT 和 Swing 其实在概念上有很多相似的地方，它们之间最大的一个不同点是：Swing 完全从跨平台的角度考虑，用 Swing 实现的组件完全是用 Java 写的，没有用到和操作系统相关的组件。但 SWT 采用了更聪明的方式，如果操作系统有相关的组件，那么 SWT 只会采用 JNI 的方式调用相关的组件，如果操作系统不提供组件才会考虑用 Java 来构建。

4.4.1 在 SWT 的组件上加入 Swing 的组件

在项目开发中，如果用 Swing 开发了前期的工作，又不想用 SWT 重写这些代码，该怎么做呢？SWT 提供了完美的解决方案，用户可以把用 Swing 编写出来的组件放在 SWT 的组件之上，不需要放弃以前的代码（但可能会牺牲界面的风格和效率）。

在 SWT 的组件中可以添加 Swing 组件。SWT 的组件只是作为 Swing 组件的一个父容器，要在 SWT 中加入 Swing 组件的步骤如下。

（1）添加 SWT 的组件。

（2）在 SWT 的组件上添加 Swing 的 Frame。

（3）在 Frame 上添加其他 Swing 的组件。

例程 4-3 展示了如何在 SWT 中添加 Swing 的 Table 组件。

例程 4-3 SwingTableInSwt.java

```
package com.free.helloworld;

import java.awt.BorderLayout;
import java.awt.Frame;
import java.awt.Panel;
import java.util.Vector;

import javax.swing.JRootPane;
import javax.swing.JScrollPane;
import javax.swing.JTable;

import org.eclipse.swt.SWT;
import org.eclipse.swt.awt.SWT_AWT;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class SwingTableInSwt {

    public static void main(String[] args) {
        final Display display = new Display();
        final Shell shell = new Shell(display);
        shell.setLayout(new FillLayout());
```

```
//新建SWT组件
Composite composite = new Composite(shell, SWT.NO_BACKGROUND
    | SWT.EMBEDDED);

try {
  System.setProperty("sun.awt.noerasebackground", "true");
} catch (NoSuchMethodError error) {
}

//在SWT组件上新建一个Frame容器
Frame frame = SWT_AWT.new_Frame(composite);
Panel panel = new Panel(new BorderLayout()) {
  public void update(java.awt.Graphics g) {

    paint(g);
  }
};
//在容器中添加了容器
frame.add(panel);
JRootPane root = new JRootPane();
panel.add(root);
java.awt.Container contentPane = root.getContentPane();

int nrows = 1000, ncolumns = 10;
Vector rows = new Vector();
for (int i = 0; i < nrows; i++) {
  Vector row = new Vector();
  for (int j = 0; j < ncolumns; j++) {
    row.addElement("Item " + i + "-" + j);
  }
  rows.addElement(row);
}
Vector columns = new Vector();
for (int i = 0; i < ncolumns; i++) {
  columns.addElement("Column " + i);
}
JTable table = new JTable(rows, columns);
table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
table.createDefaultColumnsFromModel();
JScrollPane scrollPane = new JScrollPane(table);
contentPane.setLayout(new BorderLayout());
//添加相应的组件
contentPane.add(scrollPane);

shell.open();
while (!shell.isDisposed()) {
  if (!display.readAndDispatch())
    display.sleep();
}
display.dispose();
```

```
        }
    }
```

4.4.2 程序运行效果

在上例中，SWT 组件添加了一个 Swing 的 JTable 组件，程序运行效果如图 4-9 所示。

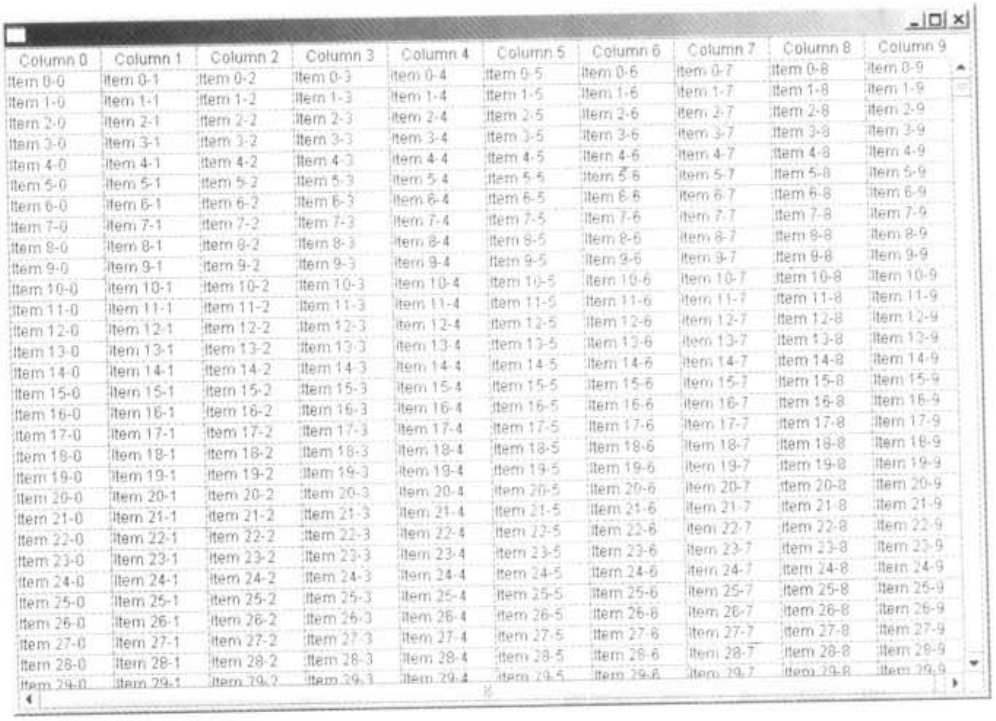


图 4-9 SWT 组件上加入 Swing 的组件

4.5 小 结

本章通过 3 个例子，介绍了 SWT 应用程序的建立、JFace 应用程序的建立和如何在 SWT 的控件上加入 Swing 的组件。本章读者应该了解如下几个方面的内容。

- 如何建立 SWT 应用程序。
- 如何建立 JFace 应用程序。
- 如何把基于 Swing 的应用移植到 SWT 中。

第 5 章　SWT 常用组件

SWT 提供了基本的标准窗口组件元素，本章主要介绍窗口组件的基类“Control”和几个 SWT 组件。

本章包括如下几个方面的内容。

- Control 类
- Button 组件
- Label 组件
- Text 组件
- Combo 组件
- List 组件

5.1　Control 类

在介绍组件之前，有必要介绍一下 Control 类。Control 类是一个抽象类，它是所有窗口组件（即在 Windows 中能获得句柄的部件）的基类。

5.1.1　Control 类的继承关系

Control 类是为继承而设计的，所有窗口组件都继承于 Control 类，如图 5-1 所示。

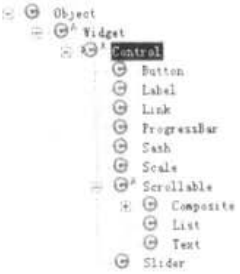


图 5-1　Control 类的继承关系

5.1.2　Control 类的常用方法

Control 类提供了窗口组件中的常用方法，所有的窗口组件都可以调用 Control 类的方法，常用方法如下。

（1）setBounds (int x, int y, int width, int height)

解释：设定窗口组件的位置，参数（x,y）为窗口组件左上角顶点的相对于父窗口坐标，（width,height）为窗口的宽度和高度。

示例：button.setBounds(40, 50, 100, 30)。

（2）setEnabled (boolean enabled)

解释：设定窗口是否可用，参数 enabled 为 true 表示窗口可用，为 false 表示窗口禁用。

示例：button.setEnabled(false)。

（3）setVisible (boolean visible)

解释：设定窗口是否可显示，参数 visible 为 true 表示窗口可显示，为 false 表示窗口不可显示。

示例：button.setVisible(false)。

（4）setToolTipText (String string)

解释：设定鼠标指向窗口时的提示信息，参数 string 为提示信息的内容。

示例：button.setToolTipText("very good")。

（5）setFont (Font font)

解释：设定窗口文字的字体，参数 font 为字体对象。

示例：button. setFont (font)。

（6）setForeground (Color color)

解释：设定窗口的前景色，参数 color 为颜色对象。

示例：button. setForeground (color)。

（7）setBackground (Color color)

解释：设定窗口的背景色，参数 color 为颜色对象。

示例：button. setBackground (color)。

（8）setCursor (Cursor cursor)

解释：设定窗口的光标形状，参数 cursor 为光标对象。

示例：button.setCursor(new Cursor(null,SWT.CURSOR_WAIT));

（9）Control (Composite parent, int style)

解释：窗口组件中的构造函数一般会调用 Control 类的构造函数，参数 parent 为当前构建的窗口的父窗口，style 为当前构建窗口的风格（默认可以指定为 SWT.NONE）。

示例：Button button = new Button(shell, SWT.NONE)。

另外，Control 类还实现了一些和窗口有关的方法，例如 createWidget 和 createHandle 等，这些方法直接和操作系统相关，有兴趣的读者可以继续研究。

5.2 Button 组件

在所有 SWT 组件中，Button 几乎是最常用的，它表示 SWT 中的按钮。Button 组件具体的介绍如下。

5.2.1 Button 组件实例

Button（按钮）是 SWT 组件常用的一种。在组件中添加一个按钮很简单，只需要指定按钮的父组件和相应的风格即可，例如："Button button = new Button(shell, SWT.PUSH)" 语句在

shell 组件中添加了一个普通的按钮。

另外，添加一个按钮一般来说会指定按钮的位置（如果未指定布局信息）和按钮的显示标签，如例程 5-1 所示。

例程 5-1 HelloWorldButton.java

```
package com.free.normal;

import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class HelloWorldButton {
      public HelloWorldButton() {
        Display display = new Display();
        Shell shell = new Shell(display);

        //指定父组件和按钮风格
        Button button = new Button(shell, SWT.PUSH);
        //指定按钮的位置
        button.setBounds(40, 50, 100, 30);
        //指定按钮的显示标签
        button.setText("Click Me");

        shell.setSize(200, 200);
        shell.open();

        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
            display.sleep();
          }
        }

        display.dispose();
      }

      public static void main(String[] args) {
        new HelloWorldButton();
      }
    }
```

上例中展示了如何在一个窗口中添加一个按钮，程序运行效果如图 5-2 所示。



图 5-2 按钮组件

5.2.2 Button 组件的风格

用户可以通过“org.eclipse.swt.widgets.Button”新建一个 SWT 的“Button”按钮，新建按钮可以指定如下不同风格的参数。

- SWT.PUSH：PUSH 按钮（普通按钮）。
- SWT.CHECK：复选框按钮。
- SWT.RADIO：单选按钮。
- SWT.TOGGLE：TOGGLE 按钮（带状态的普通按钮）。
- SWT.ARROW：箭头按钮。
- SWT.FLAT：扁平按钮。
- SWT.BORDER：带边框按钮。

其中，SWT.FLAT、SWT.BORDER 和其他风格可以同时存在。按钮风格显示如表 5-1 所示。

表 5-1 按钮风格

SWT 标识	显 示 效 果
SWT.PUSH	One
SWT.CHECK	One
SWT.RADIO	One
SWT.TOGGLE	One
SWT.ARROW	
SWT.FLAT	One
SWT.BORDER	One

5.2.3 Button 组件的对齐方式

按钮中的文字可以设置对齐方式。SWT 中按钮的对齐方式有 3 种，左对齐、右对齐和居中，另外，当为箭头按钮时，可以设置箭头向上或向下，如表 5-2 所示。

表 5-2 对齐方式

SWT 标识	对 齐 方 式	显 示 效 果
SWT.LEFT	左对齐	One
SWT.CENTER	居中对齐	One
SWT.RIGHT	右对齐	One
SWT.UP	箭头向上	
SWT.DOWN	箭头向下	

5.2.4 加入图片及按钮组合属性

用户不但可以设置按钮风格、对齐方式和状态，还能通过 setImage 方法设置按钮的图片。另外，可以同时组合这些风格、对齐方式和状态，使按钮符合用户的需求，例如风格

“SWT.BORDER | SWT.RADIO”可以设置按钮为带边框的单选按钮，如图 5-3 所示。

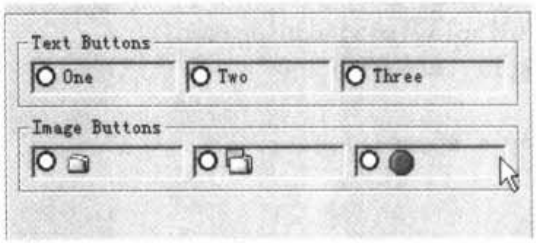


图 5-3　按钮的显示效果

5.3　Label 组件

Lable 是 SWT 中的标签组件，它是最基本组件之一，具体的介绍如下。

5.3.1　Label 组件实例

Lable（标签）是 SWT 组件常用的组件之一。在组件中添加一个标签很简单，只需要指定按钮的父组件和相应的风格即可，例如“Label label = new Label(shell, SWT.SEPARATOR | SWT.VERTICAL)”语句在 shell 组件中添加了一个标签。

标签可以作为显示文本的组件，也可以作为分隔符，如例程 5-2 所示。

例程 5-2　HelloWorldLabel.java

```
package com.free.normal;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;

public class HelloWorldLabel {

    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setLayout(new FillLayout());
        Label label1 = new Label(shell, SWT.WRAP);
        label1.setText("very good!");

        new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL);
        new Label(shell, SWT.SEPARATOR | SWT.VERTICAL);

        Label label2 = new Label(shell, SWT.NONE);
        label2.setText("gan.shu.man");

        shell.setSize(200, 70);
```

```
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch())
            display.sleep();
        }
        display.dispose();
      }
    }
```

上例窗口中添加了 4 个标签，并为每个标签设置了不同的显示风格，程序运行效果如图 5-4 所示。

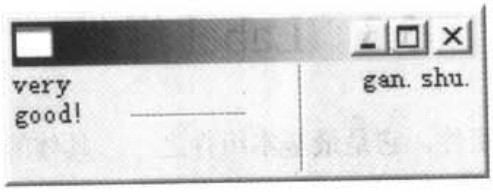


图 5-4 标签组件

5.3.2 Label 组件的风格和对齐方式

用户可以通过“org.eclipse.swt.widgets.Label”新建标签对象。标签可以有不同的风格，如表 5-3 所示。

表 5-3 标签风格

SWT 标识	含 义	显 示 效 果
SWT.SEPARATOR	标签为分隔符	
SWT.WRAP	标签文字可换行	very good!
SWT.BORDER	标签带边框	One

另外当标签风格为“SWT.SEPARATOR”时，可以设置如表 5-4 所示的标签风格。

表 5-4 分隔符样式

SWT 标识	含 义	显 示 效 果
SWT.SEPARATOR \| SWT.HORIZONTAL	水平分隔符	
SWT.SEPARATOR \| SWT.VERTICAL	垂直分隔符	
SWT.SEPARATOR \| SWT.SHADOW_IN	无阴影分隔符	
SWT.SEPARATOR \| SWT.SHADOW_OUT	有阴影分隔符	

分隔符阴影和分隔符走向属性可以同时存在，在 SWT 中有很多组合的属性，这些属性可以通过计算得到，例如水平无阴影的分隔符可用表达式“SWT.SEPARATOR | SWT.HORIZONTAL | SWT.SHADOW_IN”得到。

当标签不是分隔符时，标签文字可以设置对齐方式，如表 5-5 所示。

表 5-5 标签对齐方式

SWT 标识	对 齐 方 式	显 示 效 果
SWT.LEFT	左对齐	One
SWT.CENTER	居中对齐	One
SWT.RIGHT	右对齐	One

5.3.3 加入图片及组合属性

用户不但可以设置标签的风格、对齐方式，还能通过 setImage 设置标签的图片。另外，可以同时组合这些风格、对齐方式，使标签符合用户的需求，例如风格“SWT.BORDER | SWT.WRAP”可以设置标签为带边框的可换行标签，如图 5-5 所示。

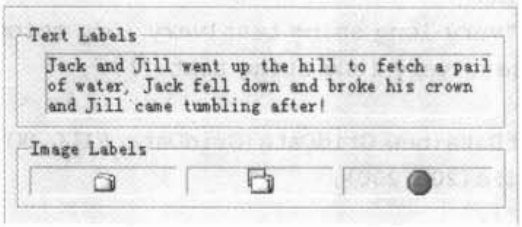


图 5-5 标签的显示效果

5.4 Text 组件

Text 是 SWT 中的输入文本框组件，具体的介绍如下。

5.4.1 Text 组件实例

Text 组件是 SWT 中的文本框，在组件中添加一个文本框很简单，只需要指定文本框的父组件和相应的风格即可，例如“Text t = new Text(shell, SWT.MULTI | SWT.BORDER | SWT.WRAP | SWT.V_SCROLL)”语句在 shell 组件中添加了一个文本框。

文本框有多种显示风格，如例程 5-3 所示。

例程 5-3 HelloWorldText.java

```
package com.free.normal;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class HelloWorldText {
```

```
public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setLayout(new GridLayout(1, false));
    // 添加单行文本框
    new Text(shell, SWT.BORDER);
    // 添加右对齐单行文本框
    new Text(shell, SWT.RIGHT | SWT.BORDER);
    // 添加以密码形式显示的文本框
    new Text(shell, SWT.PASSWORD | SWT.BORDER);
    // 添加只读文本框
    new Text(shell, SWT.READ_ONLY | SWT.BORDER).setText("Read Only");
    // 添加多行显示文本框
    Text t = new Text(shell, SWT.MULTI | SWT.BORDER | SWT.WRAP | SWT.V_SCROLL);
    //给文本属性赋值
    t.setText("very long sting test!very long sting test!very long sting
            test!very long sting test!");

    t.setLayoutData(new GridData(GridData.FILL_BOTH));
    shell.setSize(200,200);
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch()) {
        display.sleep();
      }
     }
     display.dispose();
   }
}
```

窗口中添加了 5 个不同风格的文本框，并为每个文本框设置了不同的显示风格，程序运行效果如图 5-6 所示。



图 5-6　文本框组件

5.4.2　Text 组件的风格

用户可以通过“org.eclipse.swt.widgets.Text”新建文本框对象，文本框可以有不同的风格，如表 5-6 所示。

表 5-6　文本框风格

SWT 标识	含　义	显 示 效 果
SWT.SINGLE	单行文本框	long strir
SWT.MULTI	多行文本框	very long sting test!very long sting

单行文本框和多行文本框标识是两个互斥的标识，另外，在单行文本框和多行文本框上可以添加其他的一些风格，例如“SWT.MULTI | SWT.READ_ONLY | SWT.BORDER”表示多行、只读而且带边框的文本框，可组合标识如表 5-7 所示。

表 5-7　组合文本框风格

SWT 标识	含　义	显 示 效 果
SWT.READ_ONLY	只读文本框	Read Only
SWT.BORDER	带边框文本框	
SWT.MULTI \| SWT.V_SCROLL	垂直滚动带文本框	very long sting test!very long
SWT.MULTI \| SWT.H_SCROLL	水平滚动带文本框	very long sting test!ver

另外垂直滚动条和水平滚动条只能在多行文本框时才有效。

5.5　Combo 组件

Combo 是下拉列表框组件，具体的介绍如下。

5.5.1　Combo 组件实例

Combo 组件是 SWT 中的下拉列表框，用户可以通过“Combo combo = new Combo(shell, SWT.DROP_DOWN)”在 shell 组件上添加下拉列表框，另外，可以通过“combo.setItems (ITEMS)”设置下拉列表框的下拉列表，其中“ITEMS”是 String 的数组。下拉列表框有几种显示风格，代码如例程 5-4 所示。

例程 5-4　HelloWorldCombo.java

```
package com.free.normal;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class HelloWorldCombo {

    //下拉列表项
     private static final String[] ITEMS = { "Alpha", "Bravo", "Charlie", "Delta",
```

```
    "Echo", "Foxtrot", "Golf", "Hotel", "India", "Juliet", "Kilo", "Lima", "Mike",
    "November", "Oscar", "Papa", "Quebec", "Romeo", "Sierra", "Tango", "Uniform",
    "Victor", "Whiskey", "X-Ray", "Yankee", "Zulu"
  };

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setLayout(new RowLayout());
    //添加下拉按钮风格的下拉列表框
    Combo combo = new Combo(shell, SWT.DROP_DOWN);
    //设置下拉列表项
    combo.setItems(ITEMS);

    //添加只读风格的下拉列表框
    Combo readOnly = new Combo(shell, SWT.DROP_DOWN | SWT.READ_ONLY);
    //设置下拉列表项
    readOnly.setItems(ITEMS);

    //添加无下拉按钮风格的下拉列表框
    Combo simple = new Combo(shell, SWT.SIMPLE);
    //设置下拉列表项
    simple.setItems(ITEMS);
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch()) {
        display.sleep();
      }
    }
    display.dispose();
  }
}
```

窗口中添加了 3 个不同风格的下拉列表框，并且设置了下拉列表框的显示风格，程序运行效果如图 5-7 所示。



图 5-7　下拉列表框组件

上例中只是添加了 Combo 的显示信息，通常用户会希望所选择的 Item 项关联到一个对象，当选择了某一个 Item 项后，可以直接从此 Item 项中取得所选的对象，然后操作这些对象。用户可以通过 Widget 类的“public void setData (String key, Object value)”和“public Object getData (String key)”方法实现此功能。

Combo 是 Widget 的子类，当初始化 Combo 时，可以通过 setData 方法把 Item 项的字符串和相应的对象关联起来，当选择此项时再通过 getData 方法把当前选择项的对象取出来。Widget 类中通过一个对象数据保存用户设置的对象的引用。



Widget 是所有窗口组件的父类，组件如果支持多项数据显示，并可以选择这些数据都可以通过 getData 和 setData 方法获得组件关联的对象，而且 Widget 类还提供了 getData 和 setData 其他方式的实现。

5.5.2　Combo 组件的风格

用户可以通过“org.eclipse.swt.widgets.Combo”新建下拉列表框对象。下拉列表框可以有不同的风格，如表 5-8 所示。

表 5-8　下拉列表框风格

SWT 标识	含　义	显 示 效 果
SWT.DROP_DOWN	下拉按钮风格的下拉列表框	Line 3
SWT.SIMPLE	无下拉按钮风格的下拉列表框	Line 3 Line 1 Line 2 Line 3

另外，下拉列表框风格为“SWT.DROP_DOWN”时可以设置它为只读，即“SWT.DROP_DOWN | SWT.READ_ONLY”。

5.6　List 组件

List 是 SWT 中的列表框组件，具体的介绍如下。

5.6.1　List 组件实例

List 组件是 SWT 中的列表框，用户可以通过“List single = new List(shell, SWT.BORDER | SWT.SINGLE | SWT.V_SCROLL)”在 shell 组件上添加列表框，另外，可以通过“setItems(ITEMS)”设置下拉列表框的下拉列表，其中“ITEMS”是 String 的数组。列表框有几种显示风格，代码如例程 5-5 所示。

例程 5-5　HelloWorldList.java

```
package com.free.normal;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.FillLayout;
```

```
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.List;
import org.eclipse.swt.widgets.Shell;

public class HelloWorldList {

    //列表项
     private static final String[] ITEMS = { "Alpha", "Bravo", "Charlie",
       "Delta","Echo", "Foxtrot", "Golf", "Hotel", "India", "Juliet", "Kilo",
       "Lima", "Mike", "November", "Oscar", "Papa", "Quebec", "Romeo",
       "Sierra", "Tango", "Uniform","Victor", "Whiskey", "X-Ray", "Yankee",
       "Zulu"
     };
     public static void main(String[] args) {
       Display display = new Display();
       Shell shell = new Shell(display);
       shell.setLayout(new FillLayout());
       //添加只能单选的列表框
       List single = new List(shell, SWT.BORDER | SWT.SINGLE | SWT.V_SCROLL);
       //添加列表项
       for (int i = 0, n = ITEMS.length; i < n; i++) {
         single.add(ITEMS[i]);
       }
       //选择第5项
       single.select(4);
       //添加可多选的列表框
       List multi = new List(shell, SWT.BORDER | SWT.MULTI | SWT.V_SCROLL);
      //添加列表项
       multi.setItems(ITEMS);
       //选择第10项到第12项
       multi.select(9, 11);
       shell.open();
       while (!shell.isDisposed()) {
         if (!display.readAndDispatch()) {
           display.sleep();
         }
       }
       display.dispose();
     }
    }
```

上例窗口中添加了两个列表框，一个为单选列表框，一个为多选列表框，程序运行效果如图 5-8 所示。

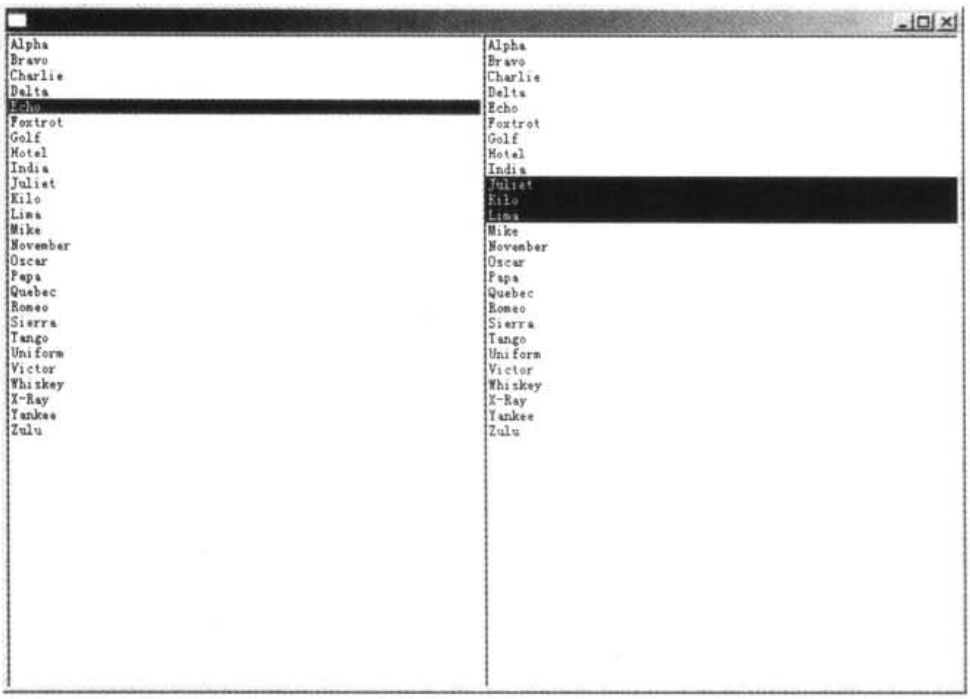


图 5-8　列表框组件

5.6.2　List 组件的风格

用户可以通过“org.eclipse.swt.widgets.List”新建列表框对象。列表框可以有不同的风格，如表 5-8 所示。

表 5-9　列表框风格

SWT 标识	含　义
SWT.SINGLE	单选列表框
SWT.MULTI	多选列表框
SWT.H_SCROLL	带水平滚动条列表框
SWT.V_SCROLL	带垂直滚动条列表框
SWT.BORDER	带边框列表框

其中，SWT.H_SCROLL、SWT.V_SCROLL 和 SWT.BORDER 可以和 SWT.SINGLE、SWT.MULTI 风格组合使用。

5.7　小　　结

本章介绍了 SWT 中窗口组件的父类“Control”，以及继承于 Control 类的几个常用组件。Control 类中实现了一些基于窗口组件的基本方法，另外不同的子类可能对一些方法提供不同的实现方式。通过学习本章读者应该了解 Control 类和常用组件的实现。

第 6 章　SWT 事件

在窗口系统中，程序一般都是以事件驱动的。SWT 的 Control 类实现了一些事件监听的注册方法，其子类都可以通过这些方法注册鼠标和键盘的监听器，通过实现这些监听器的接口，窗口组件能响应相应的鼠标和键盘事件。本章将介绍一些常用的事件，以及如何监听这些事件。

本章主要包括如下几个部分。

- 鼠标事件
- 键盘事件
- 焦点事件
- 窗口控制事件
- 其他常用事件

6.1　鼠 标 事 件

在窗口系统中，鼠标基本上是必备的设备。一般来说，窗口中鼠标有鼠标单击、鼠标双击、鼠标进入窗口、鼠标退出窗口及鼠标移动等事件。

在 SWT 中，事件的响应是通过相应接口实现的，每个组件可以添加相应的事件响应实例来监听事件。例如“button.addMouseListener(listener)”，表示在 button 组件上添加鼠标的相应响应事件实例，其中，listener 为实现监听器对象，下面就具体的鼠标事件进行介绍。

6.1.1　MouseListener 接口

在 SWT 中，通过实现 MouseListener 接口来响应鼠标的按下、松开及双击事件，MouseListener 接口如例程 6-1 所示。

例程 6-1　MouseListener.java

```
package org.eclipse.swt.events;

import org.eclipse.swt.internal.SWTEventListener;

public interface MouseListener extends SWTEventListener {

public void mouseDoubleClick(MouseEvent e);

public void mouseDown(MouseEvent e);

public void mouseUp(MouseEvent e);
}
```

其中，mouseDoubleClick 表示鼠标双击事件响应方法，mouseDown 表示鼠标键按下事件的响应方法，mouseUp 表示鼠标键起来事件的响应方法，MouseEvent 为系统传入的鼠标事件的参数，MouseEvent 中的 button 属性表示鼠标的按钮值，例如 e.button 等于 1，表示鼠标左键按下，按钮值对应鼠标按钮如表 6-1 所示。

表 6-1　鼠标按钮对应值

鼠 标 按 钮	值
鼠标左键	1
鼠标中键	2
鼠标右键	3

6.1.2　MouseMoveListener 接口

在 SWT 中，通过实现 MouseMoveListener 接口来响应鼠标的移动事件，MouseMoveListener 接口如例程 6-2 所示。

例程 6-2　MouseMoveListener.java

```
package org.eclipse.swt.events;

import org.eclipse.swt.internal.SWTEventListener;

public interface MouseMoveListener extends SWTEventListener {

public void mouseMove(MouseEvent e);
}
```

MouseMoveListener 只有 mouseMove 方法，用来响应窗口中鼠标移动事件。

6.1.3　MouseTrackListener 接口

在 SWT 中，通过实现 MouseTrackListener 接口来响应鼠标进入窗口、鼠标退出窗口和鼠标停放在窗口上的事件，MouseTrackListener 接口如例程 6-3 所示。

例程 6-3　MouseTrackListener.java

```
package org.eclipse.swt.events;

import org.eclipse.swt.internal.SWTEventListener;

public interface MouseTrackListener extends SWTEventListener {

public void mouseEnter(MouseEvent e);

public void mouseExit(MouseEvent e);

public void mouseHover(MouseEvent e);
}
```

其中，mouseEnter 表示鼠标进入窗口事件的响应方法，mouseExit 表示鼠标退出窗口事件的响应方法，mouseHover 表示鼠标停放在窗口上事件的响应方法。

6.1.4 鼠标事件实例

为了更深入地理解鼠标事件，下面通过具体的实例演示如何响应鼠标事件。该程序只是当事件触发时简单地打印出相应信息，在具体的实例中，读者可以根据需要进行修改，代码如例程 6-4 所示。

例程 6-4 MouseEventExample.java

```
package com.free.event;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.MouseEvent;
import org.eclipse.swt.events.MouseListener;
import org.eclipse.swt.events.MouseMoveListener;
import org.eclipse.swt.events.MouseTrackListener;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;

public class MouseEventExample implements MouseListener, MouseMoveListener,
        MouseTrackListener {

    //显示信息的标签
    Label myLabel = null;

    Shell shell = null;

    public MouseEventExample() {

    }

    public void run() {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setLayout(new GridLayout());
        shell.setSize(450, 200);
        shell.setText("Mouse Event Example");

        myLabel = new Label(shell, SWT.BORDER);
        myLabel.setText("I ain't afraid of any old mouse");

    //在当前窗口上添加鼠标响应事件
        shell.addMouseListener(this);
        shell.addMouseMoveListener(this);
        shell.addMouseTrackListener(this);
```

```
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        display.dispose();
    }

    public static void main(String[] args) {

        new MouseEventExample().run();
    }

    public void mouseDoubleClick(MouseEvent e) {
        myLabel.setText("Double Click " + e.button + " at: " + e.x + "," + e.y);
        System.out.println("Double Click " + e.button + " at: " + e.x + ","
                + e.y);
    }

    public void mouseDown(MouseEvent e) {
        myLabel.setText("Button " + e.button + " Down at: " + e.x + "," + e.y);
        System.out.println("Button " + e.button + " Down at: " + e.x + ","
                + e.y);
    }

    public void mouseUp(MouseEvent e) {
        myLabel.setText("Button " + e.button + " Up at: " + e.x + "," + e.y);
        System.out.println("Button " + e.button + " Up at: " + e.x + "," + e.y);
    }

    public void mouseMove(MouseEvent e) {
        myLabel.setText("Mouse Move at: " + e.x + "," + e.y);
         System.out.println("Mouse Move at: " + e.x + "," + e.y);
    }

    public void mouseEnter(MouseEvent e) {
        myLabel.setText("Mouse Enter at: " + e.x + "," + e.y);
        System.out.println("Mouse Enter at: " + e.x + "," + e.y);
    }

    public void mouseExit(MouseEvent e) {
        myLabel.setText("Mouse Exit at: " + e.x + "," + e.y);
        System.out.println("Mouse Exit at: " + e.x + "," + e.y);
    }

    public void mouseHover(MouseEvent e) {
        myLabel.setText("Mouse Hover at: " + e.x + "," + e.y);
        System.out.println("Mouse Hover at: " + e.x + "," + e.y);
    }
}
```

MouseEventExample 类实现了 MouseListener、MouseMoveListener 和 MouseTrackListener 3 个接口，并通过“shell.addMouseListener(this);”、“shell.addMouseMoveListener(this);”和“shell.addMouseTrackListener(this);”把自己作为监听器添加到了窗口中，程序运行效果如图 6-1 所示。

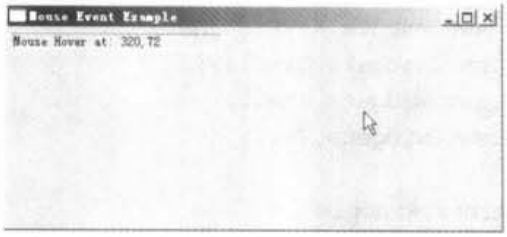


图 6-1 鼠标事件实例

6.2 键盘事件

键盘事件是最简单，也是最常用的事件。一般来说，键盘事件有两种：键按下和键松开。SWT 通过 KeyListener 响应键盘事件。

6.2.1 KeyListener 接口

在 SWT 中，通过实现 KeyListener 接口来响应键按下和松开的事件。KeyListener 接口如例程 6-5 所示。

例程 6-5 KeyListener.java

```
package org.eclipse.swt.events;

import org.eclipse.swt.internal.SWTEventListener;

public interface KeyListener extends SWTEventListener {

public void keyPressed(KeyEvent e);

public void keyReleased(KeyEvent e);
}
```

其中，keyPressed 表示键按下事件的响应方法，keyReleased 表示键松开事件的响应方法，KeyEvent 为系统传入的键盘事件的参数，用户可以通过 KeyEvent 参数找到相应的按键值。

6.2.2 键盘事件实例

为了更深入地了解键盘事件，下面通过具体的实例演示如何响应键盘事件。该程序只是当事件触发时简单地打印出相应信息，在具体的实例中，读者可以根据需要进行修改，代码如例程 6-6 所示。

例程 6-6　KeyListenerExample.java

```
package com.free.event;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.KeyEvent;
import org.eclipse.swt.events.KeyListener;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class KeyListenerExample {

      Display display;

      Shell shell;

      KeyListenerExample() {

        display = new Display();
       shell = new Shell(display);

       shell.setSize(250, 200);
       shell.setText("A KeyListener Example");
       Text text = new Text(shell, SWT.BORDER);
       text.setBounds(50 ,50 ,100, 20);

       text.addKeyListener(new KeyListener() {

         public void keyPressed(KeyEvent e) {

          System.out.println("key Pressed -"+e.character);
         }

         public void keyReleased(KeyEvent e) {
          System.out.println("key Released -"+e.character);
         }
       });
       shell.open();
       while (!shell.isDisposed()) {
        if (!display.readAndDispatch())
         display.sleep();
       }
       display.dispose();
      }

      public static void main(String[] args) {
       new KeyListenerExample();
      }

}
```

程序中通过匿名内部类实现监听器，这种方式在实例开发中会经常用到。另外可以通过KeyEvent取得按键的值，如上例的“e.character”得到按键的字符，程序运行效果如图6-2所示。

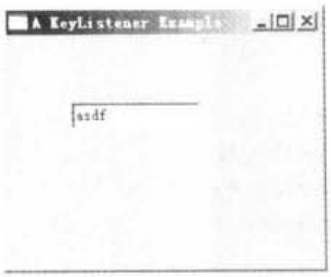


图6-2 键盘事件实例

6.3 焦点事件

在窗口系统中，当组件获得输入焦点或失去焦点时将触发相应的事件。SWT通过FocusListener监听焦点事件。

6.3.1 FocusListener接口

在SWT中，通过实现FocusListener接口来响应获得焦点和失去焦点的事件。FocusListener接口如例程6-7所示。

例程6-7 FocusListener.java

```
package org.eclipse.swt.events;


import org.eclipse.swt.internal.SWTEventListener;

public interface FocusListener extends SWTEventListener {

public void focusGained(FocusEvent e);

public void focusLost(FocusEvent e);
}
```

其中，focusGained表示组件获得焦点事件的响应方法，focusLost表示组件失去焦点事件的响应方法，FocusEvent为系统传入的焦点事件的参数，用户可以通过FocusEvent参数找到相应的组件。

6.3.2 焦点事件实例

为了更深入地理解焦点事件，下面通过具体的实例演示如何响应焦点事件。该程序只是

简单地改变当前获得焦点和失去焦点组件的显示信息，在具体的实例中，读者可以根据需要进行修改，代码如例程 6-8 所示。

例程 6-8　FocusListenerExample.java

```
package com.free.event;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class FocusListenerExample {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setLayout(new GridLayout(3, true));
    shell.setText("One Potato, Two Potato");

    //新建焦点监听器
    FocusListener listener = new FocusListener() {
      public void focusGained(FocusEvent event) {
        //获得触发事件的组件
        Button button = (Button) event.getSource();
        //焦点获得时改变显示文本
        button.setText("I'm It!");
      }

      public void focusLost(FocusEvent event) {
        //获得触发事件的组件
        Button button = (Button) event.getSource();
        //焦点获得时改变显示文本
        button.setText("Pick Me!");
      }
    };

    for (int i = 0; i < 6; i++) {
      Button button = new Button(shell, SWT.PUSH);
      button.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
      button.setText("Pick Me!");
      button.addFocusListener(listener);
    }

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch()) {
        display.sleep();
      }
    }
```

```
        display.dispose();
      }
    }
```

程序中显示了6个按钮，并为按钮添加上了焦点监听器，程序运行效果如图6-3所示。

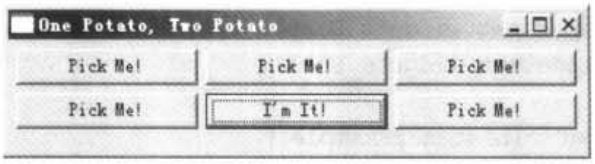


图6-3 焦点事件实例

6.4 窗口控制事件

在窗口系统中，当组件被移动或改变大小时将触发相应的事件。SWT通过ControlListener监听窗口控制事件。

6.4.1 ControlListener接口

在SWT中，通过实现ControlListener接口来响应组件被移动或改变大小的事件。ControlListener接口如例程6-9所示。

例程6-9 ControlListener.java

```
package org.eclipse.swt.events;

import org.eclipse.swt.internal.SWTEventListener;

public interface ControlListener extends SWTEventListener {

public void controlMoved(ControlEvent e);

public void controlResized(ControlEvent e);
}
```

其中，controlMoved表示组件被移动事件的响应方法，controlResized表示组件被改变大小事件的响应方法，ControlEvent为系统传入的窗口控制事件的参数，用户可以通过ControlEvent参数找到相应的组件。

6.4.2 窗口控制事件实例

为了更深入地理解窗口控制事件，下面通过具体的实例演示如何响应窗口控制事件。该程序只是简单地打印组件被移动或改变大小的信息，在具体的实例中，读者可以根据需要进行修改，代码如例程6-10所示。

例程 6-10 ControlListenerExample.java

```
package com.free.event;

import org.eclipse.swt.events.ControlEvent;
import org.eclipse.swt.events.ControlListener;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class ControlListenerExample {
  public void run() {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.addControlListener(new ControlListener() {

          public void controlMoved(ControlEvent e) {
              System.out.println("control move");
          }

          public void controlResized(ControlEvent e) {
              System.out.println("control resize");
          }

    }
     );
    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch()) {
        display.sleep();
      }
    }
    display.dispose();
  }

  public static void main(String[] args) {
    new ControlListenerExample().run();
  }
}
```

上例显示的只是一个空白的窗口，当用户移动窗口或改变窗口大小时，将会触发监听器所实现的事件。

6.5 选择组件事件

SWT 的很多组件都实现了选择组件事件的监听机制，例如按钮、菜单项的选择，当选择了相关的菜单项或组件时，将触发相应的事件。

6.5.1 SelectionListener 接口

在 SWT 中，通过实现 SelectionListener 接口来响应选择组件事件。SelectionListener 接口如例程 6-11 所示。

例程 6-11 SelectionListener.java

```
package org.eclipse.swt.events;

import org.eclipse.swt.internal.SWTEventListener;

public interface SelectionListener extends SWTEventListener {

public void widgetSelected(SelectionEvent e);

public void widgetDefaultSelected(SelectionEvent e);
}
```

其中，widgetSelected 表示组件被选择事件的响应方法，widgetDefaultSelected 表示组件默认选择事件的响应方法，SelectionEvent 为系统传入的选择事件的参数。

6.5.2 选择组件事件实例

为了更深入地理解选择组件事件，下面通过具体的实例演示如何响应选择组件事件。该程序只是简单地打印组件被移动或改变大小的信息，在具体的实例中，读者可以根据需要进行修改，代码如例程 6-12 所示。

例程 6-12 SelectonListenerExample.java

```
package com.free.event;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.DisposeEvent;
import org.eclipse.swt.events.DisposeListener;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;

public class SelectonListenerExample {

    public static void main(String[] args) {
        Display display = new Display();

        final Shell mainShell = new Shell(display);
```

```
        Button button = new Button(mainShell, SWT.PUSH);
        button.setText("Close Me!");
        button.setBounds(10, 10, 100, 30);
        // 添加选择组件事件
        button.addSelectionListener(new SelectionListener() {
            public void widgetSelected(SelectionEvent event) {
                System.out.println("select button");
                mainShell.close();
            }

            public void widgetDefaultSelected(SelectionEvent e) {
                // 不执行任何操作
            }
        });

        mainShell.open();

        while (!mainShell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }
        display.dispose();
    }
}
```

上例中只是添加了一个按钮，当选择按钮时关闭当前窗口。

6.6 其他常用事件

SWT 中为了实现特定的功能，很多组件都提供了特定事件的响应，本节将通过实例介绍 HelpListener、VerifyListener 和 ModifyListener 3 个特定的事件监听器。

6.6.1 HelpListener、VerifyListener 和 ModifyListener 监听器的功能

HelpListener 监听器通过 helpRequested(HelpEvent e)方法响应用户的帮助请求事件，当用户在组件获得焦点后按【F1】键将触发此事件。

VerifyListener 监听器通过 verifyText(VerifyEvent e)方法响应校验输入事件。此监听器只对文本输入校验，当用户输入了数据后，verifyText 方法将通过设置 VerifyEvent 中的 doit 属性判断输入是否正确，从而确定修改是否有效。doit 属性为 true 时修改有效，即“e.doit = true;”。

ModifyListener 监听器通过 modifyText(ModifyEvent e)方法响应文本被修改的事件。此监听器只对文本输入校验。



如果 VerifyListener 监听器和 ModifyListener 监听器同时存在的话，会先响应校验输入事件，如果校验成功再响应修改事件。

6.6.2 HelpListener、VerifyListener 和 ModifyListener 监听器实例

在此实例中，用户可以输入华氏温度和摄氏温度，通过监听器判断输入是否正确及计算相应的摄氏温度和华氏温度，另外，还可以按【F1】键获得当前组件的信息，代码如例程 6-13 所示。

例程 6-13 MultipleListenersExample.java

```
package com.free.event;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class MultipleListenersExample implements HelpListener, VerifyListene
r,
   ModifyListener {

 private static final double FIVE_NINTHS = 5.0 / 9.0;
 private static final double NINE_FIFTHS = 9.0 / 5.0;

 private Text fahrenheit;
 private Text celsius;
 private Label help;

 public void run() {
  Display display = new Display();
  Shell shell = new Shell(display);
  shell.setText("Temperatures");
  createContents(shell);
  shell.pack();
  shell.open();
  while (!shell.isDisposed()) {
   if (!display.readAndDispatch()) {
    display.sleep();
   }
  }
  display.dispose();
 }

 private void createContents(Shell shell) {
  shell.setLayout(new GridLayout(3, true));

  new Label(shell, SWT.LEFT).setText("Fahrenheit:");
  fahrenheit = new Text(shell, SWT.BORDER);
```

```
        GridData data = new GridData(GridData.FILL_HORIZONTAL);
        data.horizontalSpan = 2;
        fahrenheit.setLayoutData(data);

        fahrenheit.setData("Type a temperature in Fahrenheit");

        // 为华氏温度文本框添加监听器
        fahrenheit.addHelpListener(this);
        fahrenheit.addVerifyListener(this);
        fahrenheit.addModifyListener(this);

        new Label(shell, SWT.LEFT).setText("Celsius:");
        celsius = new Text(shell, SWT.BORDER);
        data = new GridData(GridData.FILL_HORIZONTAL);
        data.horizontalSpan = 2;
        celsius.setLayoutData(data);

        celsius.setData("Type a temperature in Celsius");

        //为摄氏温度文本框添加监听器
        celsius.addHelpListener(this);
        celsius.addVerifyListener(this);
        celsius.addModifyListener(this);

        help = new Label(shell, SWT.LEFT | SWT.BORDER);
        data = new GridData(GridData.FILL_HORIZONTAL);
        data.horizontalSpan = 3;
        help.setLayoutData(data);
      }

    //响应帮助事件
    public void helpRequested(HelpEvent event) {

      help.setText((String) event.widget.getData());
    }
    //响应校验事件
    public void verifyText(VerifyEvent event) {
      event.doit = false;

      char myChar = event.character;
      String text = ((Text) event.widget).getText();

      if (myChar == '-' && text.length() == 0) event.doit = true;

     if (Character.isDigit(myChar)) event.doit = true;

      if (myChar == '\b') event.doit = true;
    }

    //响应文本修改的事件
    public void modifyText(ModifyEvent event) {
```

```
        // 删除监听器，从而在modifyText过程中不会触发事件
        celsius.removeVerifyListener(this);
        celsius.removeModifyListener(this);
        fahrenheit.removeVerifyListener(this);
        fahrenheit.removeModifyListener(this);

        Text text = (Text) event.widget;

        try {
          int temp = Integer.parseInt(text.getText());

          if (text == fahrenheit) {
            celsius.setText(String.valueOf((int) (FIVE_NINTHS * (temp - 32))));
          } else {
            fahrenheit.setText(String.valueOf((int) (NINE_FIFTHS * temp + 32)));
          }
        } catch (NumberFormatException e) { /* Ignore */ }

        //添加监听器
        celsius.addVerifyListener(this);
        celsius.addModifyListener(this);
        fahrenheit.addVerifyListener(this);
        fahrenheit.addModifyListener(this);
      }

      public static void main(String[] args) {
        new MultipleListenersExample().run();
      }
    }
```

程序运行效果如图6-4所示。

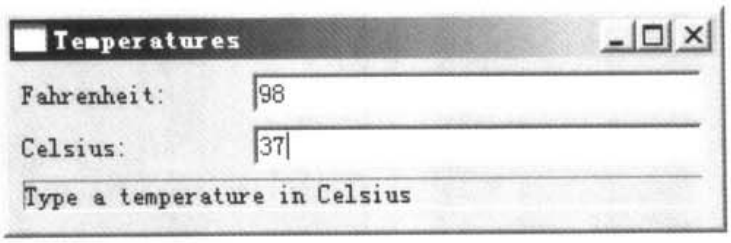


图6-4 文本监听器

注意 一般来说，监听器都有一个抽象的 Adaper 类实现监听器的方法，例如FocusAdapter实现了FocusListener的方法（方法为空）。如果读者不想实现监听器的全部方法则可以继承监听器的Adaper类，否则要实现监听器接口的所有方法。

6.7 小 结

本章通过实例形象地介绍了SWT中事件监听器的用法，但SWT中还有一些特定监听器，

例如 PaintListener 等，在此无法一一介绍，如果有需要，读者可以通过查找帮助找到这些监听器的实现，它们的实现方式和本章的实例类似。

另外，在有些场景下，组件可以通过实现不同的监听器实现相同的功能，例如选择组件事件和鼠标事件都能完成单击组件的功能，但它们代表的事件是不一样的，读者在开发的时候可以考虑选择实现的接口。

第 7 章　SWT 布局管理

在 SWT 中，如果用户想要控制窗口中元素的排列、大小和位置等信息，最好的办法是通过设置组件的布局信息，通过布局来控制组件子元素外观，本章主要介绍 SWT 中常用的几种布局方式。

本章主要内容如下所示。

- 概述
- FillLayout
- RowLayout
- GridLayout
- FormLayout
- 创建自己布局

7.1　概　　述

在 Java 中，GUI 程序开发的目标之一是跨平台，而每种类型操作系统对屏幕的定义不一样，所以 Swing 中引入了布局的概念，对子组件的位置和大小等信息进行定义。SWT 中也采用了布局方式，用户可使用布局来控制组件中元素的位置和大小等信息。

另外，如果组件没有设置布局信息，也可以通过坐标进行相对定位。在介绍 Control 类的时候，其中有一个方法“setBounds (int x, int y, int width, int height)”可以指定组件相对于父组件的位置和组件的大小。在这种方式下，父组件大小和其他信息的改变不会对当前组件有影响。复合组件常常包含多个控件，可以使用以下两种方法安排这些控件。

（1）绝对定位：为每个控件设置明确的 *X* 和 *Y* 位置（setBounds），并通过代码设置一定的宽度和高度。

（2）托管定位：每个控件的 *X*、*Y*、宽度和高度都是通过 LayoutManager 设置的。

在多数情况下，应该选择使用 LayoutManagers，因为可以很容易地调整它们来适应可变大小的 GUI。SWT 提供了一些常用的布局管理器供用户使用；在本章中，将介绍 4 种基本的布局管理器：FillLayout、RowLayout、GridLayout 和 FormLayout。在布局管理器中，每当重新设置复合组件的大小时，都需要进行定位。

布局管理器常常是专为某一个复合组件设计的。一些布局管理器只使用它们自身的参数就可以控制，而另一些布局管理器还需要其他参数（LayoutData），该参数是在设置布局管理器的复合组件中的每个控件上指定的。SWT 中常用的布局管理器有如下一些。

- FillLayout：子组件将以相同的大小填充到父组件中。
- RowLayout：子组件将在父组件上一行或几行显示（设置相应的属性值，子组件会自动换行）。

- GridLayout：网格式布局，子组件可以指定占用父组件中几个格，以及组件填充哪几个网格。
- FormLayout：可以通过 FormAttachment 以父组件或子组件的边作为相对位置，进行精确布局。

为组件添加布局信息的步骤如下。

（1）创建布局类。

（2）通过窗口组件的 setLayout 方法设置相应的布局类。

（3）设置子组件的布局信息。

当窗口组件设置了布局信息后，窗口组件显示的时候将会调用相应的布局类对窗口组件的子组件进行布局、定位和计算子组件大小的操作，从而使窗口组件以更好的方式显示在父组件中。下面将介绍 Eclipse 中提供的几种常用的布局方式。

7.2 FillLayout 布局

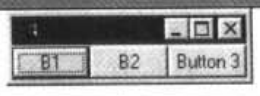
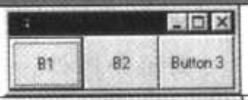
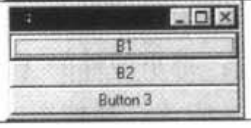
FillLayout 是非常简单的一种布局方式，它会以同样大小对父组件中的子组件进行布局，这些子组件将以一行或一列的形式排列。

一般来说，用户可以在任务栏、工具栏中放置 FillLayout 布局，通过 FillLayout 布局对子组件进行定位，也可以当子组件只有一个组件时，通过 FillLayout 布局填充整个父组件的空间。

7.2.1 FillLayout 的风格

FillLayout 布局中，可以把子组件按水平或垂直的方式进行排列，这些风格是当创建 FillLayout 实类时以参数形式指定的，如表 7-1 所示。

表 7-1 FillLayout 的风格

	初始状态	调整父窗口大小后的状态
SWT.HORIZONTAL (default)		
SWT.VERTICAL		

7.2.2 FillLayout 布局实例

FillLayout 是简单而且很常用的布局，下面通过实例展示 FillLayout 的布局效果，代码如例程 7-1 所示。

例程 7-1 FillLayoutSample.java

```
package com.free.layout;

import org.eclipse.swt.SWT;
```

```
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class FillLayoutSample {
      Display display = new Display();
      Shell shell = new Shell(display);

      public FillLayoutSample() {
        //新建FillLayout布局，设置了组件与水平方式排列
        FillLayout fillLayout = new FillLayout(SWT.HORIZONTAL);
        //指定子组件的上、下边距为多少像素
        fillLayout.marginHeight = 25;
        //指定子组件的左、右边距为多少像素
        fillLayout.marginWidth = 25;
        //指定子组件之间距离为多少像素
        fillLayout.spacing = 10;

        //设定父组件的布局方式
        shell.setLayout(fillLayout);
        Button button1 = new Button(shell, SWT.PUSH);
        button1.setText("button1");
        Button button2 = new Button(shell, SWT.PUSH);
        button2.setText("button number 2");
        Button button3 = new Button(shell, SWT.PUSH);
        button3.setText("3");

        shell.pack();
        shell.open();

        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
            display.sleep();
          }
        }

        display.dispose();
      }

      public static void main(String[] args) {
        new FillLayoutSample();
      }
    }
```

程序中通过 marginHeight、marginWidth 和 spacing 指定了边距和子组件的间距，程序运行效果如图 7-1 所示。

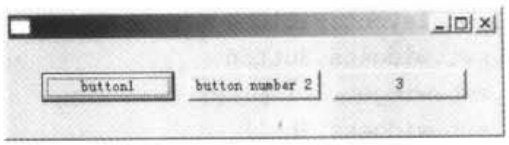


图 7-1　FillLayout 布局实例

7.3　RowLayout 布局

相对于 FillLayout 来说，RowLayout 比较灵活，功能也比较强。用户可以设置布局中子元素的大小、边距、换行及间距等属性。

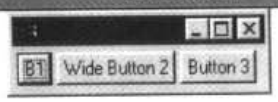
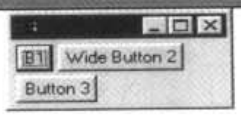
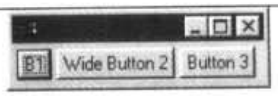
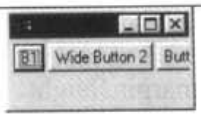
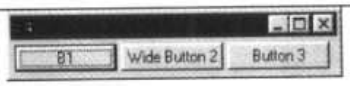
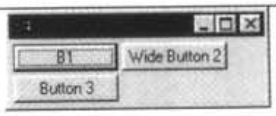
7.3.1　RowLayout 的风格

RowLayout 中可以相关的属性设定布局的风格，用户可以通过“RowLayout.属性”的方式设置 RowLayout 的布局风格，RowLayout 中常用的属性如下。

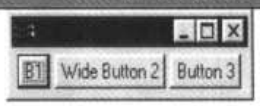
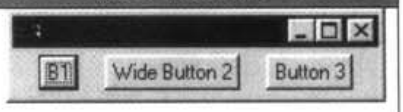
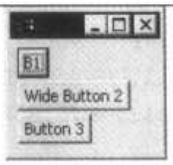
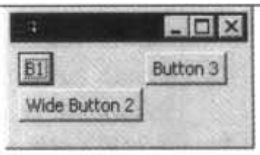
- Wrap：表示子组件是否可以换行（true 为可换行）。
- Pack：表示子组件是否为保持原有大小（true 为保持原有大小）。
- Justify：表示子组件是否根据父组件信息做调整。
- MarginLeft：表示当前组件距离父组件左边距的像素点个数。
- MarginTop：表示当前组件距离父组件上边距的像素点个数。
- MarginRight：表示当前组件距离父组件右边距的像素点个数。
- MarginBottom：表示当前组件距离父组件下边距的像素点个数。
- Spacing：表示子组件之间的间距像素点个数。

另外，RowLayout 可以通过 RowData 设置每个子组件的大小，例如“button.setLayoutData (new RowData(60, 60))”将设置 buton 的大小为（60,60），RowLayout 风格如表 7-2 所示。

表 7-2　RowLayout 风格

	初 始 状 态	调整大小后的状态
wrap = true pack = true justify = false type = SWT.HORIZONTAL (defaults)		
wrap = false (clips if not enough space)		
pack = false (all widgets are the same size)		

（续表）

	初 始 状 态	调整大小后的状态
justify = true (widgets are spread across the available space)		
type = SWT.VERTICAL (widgets are arranged vertically in columns)		

7.3.2　RowLayout 布局实例

RowLayout 是很常用的布局，而且不太复杂，下面通过实例展示 RowLayout 的布局效果，代码如例程 7-2 所示。

例程 7-2　RowLayoutExample.java

```
package com.free.layout;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.RowData;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;


public class RowLayoutExample {
    Display display;

    Shell shell;

    RowLayoutExample() {
        display = new Display();
        shell = new Shell(display);
        shell.setSize(250, 150);

        shell.setText("A RowLayout Example");
        //新建 RowLayout 布局
        RowLayout rowLayout = new RowLayout();
        //子组件保持原有大小
        rowLayout.pack = true;
        //子组件可换行
        rowLayout.wrap = true;
        //根据父组件信息调整位置
        rowLayout.justify = true;
        //左边距为 30 像素
```

```
        rowLayout.marginLeft = 30;
        //上边距为30像素
        rowLayout.marginTop = 30;

        //设定父组件RowLayout布局
        shell.setLayout(rowLayout);

        final Text t = new Text(shell, SWT.SINGLE | SWT.BORDER);
        final Button b = new Button(shell, SWT.BORDER);
        final Button b1 = new Button(shell, SWT.BORDER);
        //设置子组件大小
        b1.setLayoutData(new RowData(60, 60));

        b.setText("OK");
        b1.setText("Cancel");
        shell.open();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }

    public static void main(String[] argv) {
        new RowLayoutExample();
    }

}
```

程序中指定了边距和子组件的间距，以及子组件大小的信息，程序运行效果如图 7-2 所示。

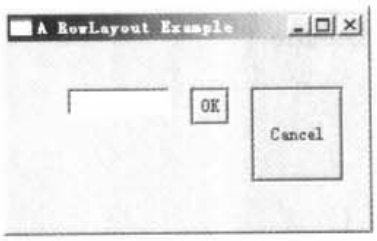


图 7-2　RowLayout 布局实例

7.4　GridLayout 布局

GridLayout 布局的功能非常强大，也是笔者常用的一种布局方式。GridLayout 是网格式布局，它把父组件分成一个表格，默认情况下每个子组件占据一个单元格的空间，每个子组件按添加到父组件的顺序排列在表格中。

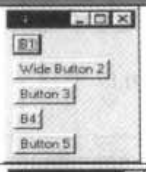
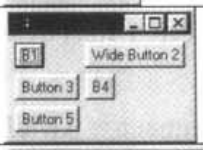
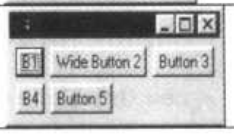
GridLayout 提供了很多的属性，可以灵活设置网格的信息。另外，GridLayout 布局提供了 GridData 类，子组件可以设置相应的 GridData，例如“dogPhoto.setLayoutData(gridData)”，GridData 可以设置每个组件当做单元格的信息。

7.4.1　GridLayout 的风格

GridLayout 类提供了 GridLayout 布局中划分网格的信息，主要通过以下几个参数进行设置。

- NumColumns：通过“gridLayout.numColumns”属性可以设置父组件中分几列显示子组件，如表 7-3 所示。

表 7-3　NumColumns 效果

列　数	显 示 效 果
numColumns = 1	
numColumns = 2	
numColumns = 3	

- MakeColumnsEqualWidth：通过“gridLayout. makeColumnsEqualWidth”属性可以设置父组件中子组件是否有相同的列宽，当 MakeColumnsEqualWidth 为 true 时表示每列的列宽相等。
- MarginLeft：表示当前组件距离父组件左边距的像素点个数。
- MarginRight：表示当前组件距离父组件右边距的像素点个数。
- MarginTop：表示当前组件距离父组件上边距的像素点个数。
- MarginBottom：表示当前组件距离父组件下边距的像素点个数。
- HorizontalSpacing：表示子组件的水平间距。
- VerticalSpacing：表示子组件的垂直间距。

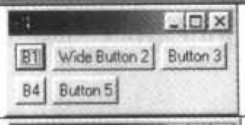
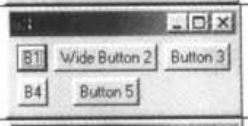
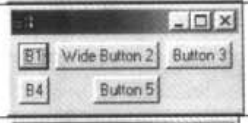
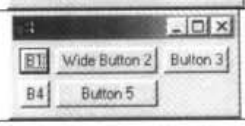
7.4.2　GridData 的相关属性

GridLayout 布局的灵活之处在于它利用网格布局数据 GridData。通过 GridData 可以设置子组件在网格中的填充方式、大小边距等信息，用户可以通过子组件的 setLayoutData 方法设置网格布局数据。

GridData 可以控制子组件在网格中的位置大小等相关显示信息。GridData 可以设置如下的一些属性。

- HorizontalAlignment：表示水平对齐方式。水平对齐方式有如下几种，如表 7-4 所示，其中“Button5”按钮显示了水平对齐的方式。

表 7-4　组件水平对齐方式

HorizontalAlignment 的值	显 示 效 果
horizontalAlignment = GridData.BEGINNING (default)	
horizontalAlignment = GridData.CENTER	
horizontalAlignment = GridData.END	
horizontalAlignment = GridData.FILL	

- VerticalAlignment：表示子组件的垂直对齐方式，值和水平方式一样。
- HorizontalIndent：表示子组件水平偏移多少像素。

此属性和“horizontalAlignment = GridData.BEGINNING”属性一起使用。下面代码设置“Button5”水平偏移 4 像素，如图 7-3 所示。

```
GridData gridData = new GridData();
gridData.horizontalIndent = 4;
button5.setLayoutData(gridData);
```

- HorizontalSpan：表示组件水平占据几个网格。

此属性非常有用，当要设置一个组件占据几个单元格时，需要设置 HorizontalSpan 属性。例如，下面代码设置“Button5”按钮水平占据两个网格，如图 7-4 所示。

```
GridData gridData = new GridData();
gridData.horizontalAlignment = GridData.FILL;
gridData.horizontalSpan = 2;
button5.setLayoutData(gridData);
```

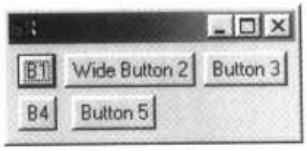


图 7-3　组件水平偏移

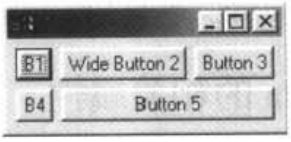


图 7-4　水平占据网格

- VerticalSpan：表示组件垂直占据几个网格。
- GrabExcessHorizontalSpace：表示当父组件大小改变时，子组件是否以水平方向抢占空间。
- GrabExcessVerticalSpace：表示当父组件大小改变时，子组件是否以垂直方向抢占空间。
- WidthHint：表示子组件的宽度为多少像素（前提是未设置其他相关属性）。
- HeightHint：表示子组件的高度为多少像素（前提是未设置其他相关属性）。

另外，GridData 可以通过构造函数指定相应的属性值，有兴趣的读者可以参考 GridData

类的构造函数。

7.4.3 GridLayout 布局实例

为了更深入地理解 GridLayout 布局，下面通过具体的实例演示如何构建一个比较复杂的布局窗口。通过本例的学习，读者可以比较好地掌握 GridLayout 布局，代码如例程 7-3 所示。

例程 7-3 ComplexGridLayoutExample.java

```
package com.free.layout;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.PaintEvent;
import org.eclipse.swt.events.PaintListener;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Canvas;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.FileDialog;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.List;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class ComplexGridLayoutExample {

    static Display display;
    static Shell shell;
    static Text dogName;
    static Combo dogBreed;
    static Canvas dogPhoto;
    static Image dogImage;
    static List categories;
    static Text ownerName;
    static Text ownerPhone;

    public static void main(String[] args) {

        display = new Display();
        shell = new Shell(display);
        shell.setText("Dog Show Entry");

        //新建GridLayout 布局
```

```
GridLayout gridLayout = new GridLayout();
//把了组件分成3列显示
gridLayout.numColumns = 3;

shell.setLayout(gridLayout);
new Label(shell, SWT.NONE).setText("Dog's Name:");
dogName = new Text(shell, SWT.SINGLE | SWT.BORDER);

//新建水平填充的GridData
GridData gridData = new GridData(GridData.HORIZONTAL_ALIGN_FILL);
//GridData的组件占两列显示
gridData.horizontalSpan = 2;

dogName.setLayoutData(gridData);

new Label(shell, SWT.NONE).setText("Breed:");

dogBreed = new Combo(shell, SWT.NONE);
dogBreed.setItems(new String [] {"Collie", "Pitbull", "Poodle", "Scottie"});
dogBreed.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_FILL));

Label label = new Label(shell, SWT.NONE);
label.setText("Categories");
label.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_CENTER));

new Label(shell, SWT.NONE).setText("Photo:");

dogPhoto = new Canvas(shell, SWT.BORDER);

//gridData两端填充
gridData = new GridData(GridData.FILL_BOTH);

//gridData最佳宽度和高度
gridData.widthHint = 80;
gridData.heightHint = 80;

//设置gridData的了组件垂直占3行
gridData.verticalSpan = 3;

dogPhoto.setLayoutData(gridData);
//添加画布的重画事件
dogPhoto.addPaintListener(new PaintListener() {
  public void paintControl(final PaintEvent event) {
     if (dogImage != null) {
       event.gc.drawImage(dogImage, 0, 0);
     }
  }
});

categories = new List(shell, SWT.MULTI | SWT.BORDER | SWT.V_SCROLL);
categories.setItems(new String [] {
```

```
    "Best of Breed", "Prettiest Female", "Handsomest Male",
    "Best Dressed", "Fluffiest Ears", "Most Colors",
    "Best Performer", "Loudest Bark", "Best Behaved",
    "Prettiest Eyes", "Most Hair", "Longest Tail",
    "Cutest Trick"});

gridData = new GridData(GridData.HORIZONTAL_ALIGN_FILL | GridData.
          VERTICAL_ALIGN_FILL);
gridData.verticalSpan = 4;
int listHeight = categories.getItemHeight() * 12;
Rectangle trim = categories.computeTrim(0, 0, 0, listHeight);
gridData.heightHint = trim.height;
categories.setLayoutData(gridData);
Button browse = new Button(shell, SWT.PUSH);
browse.setText("Browse...");
gridData = new GridData(GridData.HORIZONTAL_ALIGN_FILL);
gridData.horizontalIndent = 5;
browse.setLayoutData(gridData);

//添加按钮的选择事件
browse.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent event) {
     String fileName = new FileDialog(shell).open();
     if (fileName != null) {
       dogImage = new Image(display, fileName);
       dogPhoto.redraw();
     }
  }
});

Button delete = new Button(shell, SWT.PUSH);
delete.setText("Delete");
gridData = new GridData(GridData.HORIZONTAL_ALIGN_FILL | GridData.
          VERTICAL_ALIGN_BEGINNING);
gridData.horizontalIndent = 5;
delete.setLayoutData(gridData);

delete.addSelectionListener(new SelectionAdapter() {
  public void widgetSelected(SelectionEvent event) {
     if (dogImage != null) {
       dogImage.dispose();
       dogImage = null;
       dogPhoto.redraw();
     }
  }
});

Group ownerInfo = new Group(shell, SWT.NONE);
ownerInfo.setText("Owner Info");
gridLayout = new GridLayout();
gridLayout.numColumns = 2;
```

```
        ownerInfo.setLayout(gridLayout);
        gridData = new GridData(GridData.HORIZONTAL_ALIGN_FILL);
        gridData.horizontalSpan = 2;
        ownerInfo.setLayoutData(gridData);

        new Label(ownerInfo, SWT.NONE).setText("Name:");
        ownerName = new Text(ownerInfo, SWT.SINGLE | SWT.BORDER);
        ownerName.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

        new Label(ownerInfo, SWT.NONE).setText("Phone:");
        ownerPhone = new Text(ownerInfo, SWT.SINGLE | SWT.BORDER);
        ownerPhone.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

        Button enter = new Button(shell, SWT.PUSH);
        enter.setText("Enter");
        gridData = new GridData(GridData.HORIZONTAL_ALIGN_END);
        gridData.horizontalSpan = 3;
        enter.setLayoutData(gridData);

        enter.addSelectionListener(new SelectionAdapter() {
          public void widgetSelected(SelectionEvent event) {
            System.out.println("\nDog Name: " + dogName.getText());
            System.out.println("Dog Breed: " + dogBreed.getText());
            System.out.println("Owner Name: " + ownerName.getText());
            System.out.println("Owner Phone: " + ownerPhone.getText());
            System.out.println("Categories:");
            String cats[] = categories.getSelection();
            for (int i = 0; i < cats.length; i++) {
              System.out.println("\t" + cats[i]);
            }
          }
        });

        shell.pack();
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) display.sleep();
        }

        if (dogImage != null) {
          dogImage.dispose();
        }
      }
    }
```

程序中构建了一个比较复杂的窗口，设置了 GridLayout 的布局信息和相关的 GridData 网格数据信息，程序运行效果如图 7-5 所示。

注意 每个 GridData 都必须有单独的实例，几个子组件不能够共用同一个 GridData（即使几个组件有相同属性的 GridData 也必须新建几个实例）。

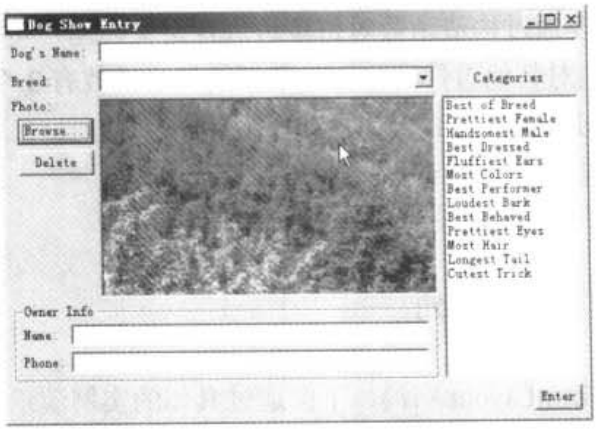


图 7-5 GridLayout 布局实例

7.5 FormLayout 布局

FormLayout 是一种比较灵活的布局方式，它的功能很强大。FormLayout 中能指定每个子组件的 FormData，每个 FormData 表示当前组件的布局信息。另外，FormData 能指定子组件每条边对应的附加信息（FormAttachment）。FormLayout 主要利用组件的每条边对应其他组件的位置进行布局。

7.5.1 FormLayout 布局信息

FormLayout 主要通过 FormData 设置子组件的位置和边界信息来对父组件元素进行布局，另外，FormLayout 布局中也可以设置父组件相对于子组件的边距值。

1．FormLayout 属性

- MarginLeft：表示当前组件距离父组件左边距的像素点个数。
- MarginRight：表示当前组件距离父组件右边距的像素点个数。
- MarginTop：表示当前组件距离父组件上边距的像素点个数。
- MarginBottom：表示当前组件距离父组件下边距的像素点个数。

2．FormData 属性

- Top：指定子组件的上面边的 FormAttachment。
- bottom：指定子组件的下面边的 FormAttachment。
- Left：指定子组件的左面边的 FormAttachment。
- Right：指定子组件的右面边的 FormAttachment。
- Width：指定子组件的宽度（如果未与其他属性冲突时有效）。
- Height：指定子组件的高度（如果未与其他属性冲突时有效）。

用户可通过属性设置组件边的附加信息，例如“formData.top = new FormAttachment(0,60)”指定了 formData 对象上面边的信息。通过 FormAttachment 可以定义组件的某条特定的边，用户可以指定一个 FormData 对象的 4 条边的信息，但为了有效地用 FormLayout 对组件进行布局，最少指定两条边以上最好。

FormAttachment 主要通过构造函数对每条边进行定义，用户可以定义某条边相对于父组件的位置，也可以定义相对其他组件的位置。常用的构造函数有 3 个参数。

- numerator：相对于组件的百分比。
- offset：相对于组件的偏移量。
- control：组件（如果未指定则默认为父组件）。

7.5.2 FormLayout 布局实例

为了更深入地理解 FormLayout 布局，下面通过具体的实例演示如何构建一个比较复杂的布局窗口。本例和 GridLaylout 布局实例实现的效果一样，但解决方案不一样，在此采用 FormLayout 布局方式，代码如例程 7-4 所示。

例程 7-4 ComplexFormLayoutExample.java

```
package com.free.layout;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.PaintEvent;
import org.eclipse.swt.events.PaintListener;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.FormAttachment;
import org.eclipse.swt.layout.FormData;
import org.eclipse.swt.layout.FormLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Canvas;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.FileDialog;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.List;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;


public class ComplexFormLayoutExample {
    static Display display;

    static Shell shell;

    static Image dogImage;

    static Text dogNameText;

    static Combo dogBreedCombo;

    static Canvas dogPhoto;
```

```
static List categories;

static Text nameText;

static Text phoneText;

public static void main(String[] args) {
    display = new Display();
    shell = new Shell(display);
    FormLayout layout = new FormLayout();
    layout.marginWidth = 5;
    layout.marginHeight = 5;
    shell.setLayout(layout);
    shell.setText("Dog Show Entry");

    Group ownerInfo = new Group(shell, SWT.NONE);
    ownerInfo.setText("Owner Info");
    FormLayout ownerLayout = new FormLayout();
    ownerLayout.marginWidth = 5;
    ownerLayout.marginHeight = 5;
    ownerInfo.setLayout(ownerLayout);

    Label dogName = new Label(shell, SWT.NONE);
    dogName.setText("Dog's Name:");
    dogNameText = new Text(shell, SWT.SINGLE | SWT.BORDER);
    Label dogBreed = new Label(shell, SWT.NONE);
    dogBreed.setText("Breed:");
    dogBreedCombo = new Combo(shell, SWT.NONE);
    dogBreedCombo.setItems(new String[] { "Collie", "Pitbull", "Poodle",
            "Scottie" });
    Label photo = new Label(shell, SWT.NONE);
    photo.setText("Photo:");
    dogPhoto = new Canvas(shell, SWT.BORDER);
    Button browse = new Button(shell, SWT.PUSH);
    browse.setText("Browse...");
    Button delete = new Button(shell, SWT.PUSH);
    delete.setText("Delete");
    Label cats = new Label(shell, SWT.NONE);
    cats.setText("Categories");
    categories = new List(shell, SWT.MULTI | SWT.BORDER | SWT.V_SCROLL
            | SWT.H_SCROLL);
    categories.setItems(new String[] { "Best of Breed", "Prettiest Female",
            "Handsomest Male", "Best Dressed", "Fluffiest Ears",
            "Most Colors", "Best Performer", "Loudest Bark",
            "Best Behaved", "Prettiest Eyes", "Most Hair", "Longest Tail",
            "Cutest Trick" });
    Button enter = new Button(shell, SWT.PUSH);
    enter.setText("Enter");

    FormData data = new FormData();
```

```
data.top = new FormAttachment(dogNameText, 0, SWT.CENTER);
dogName.setLayoutData(data);

data = new FormData();
data.left = new FormAttachment(dogName, 5);
data.right = new FormAttachment(100, 0);
dogNameText.setLayoutData(data);

data = new FormData();
data.top = new FormAttachment(dogBreedCombo, 0, SWT.CENTER);
dogBreed.setLayoutData(data);

data = new FormData();
data.top = new FormAttachment(dogNameText, 5);
data.left = new FormAttachment(dogNameText, 0, SWT.LEFT);
data.right = new FormAttachment(categories, -5);
dogBreedCombo.setLayoutData(data);

data = new FormData(80, 80);
data.top = new FormAttachment(dogBreedCombo, 5);
data.left = new FormAttachment(dogNameText, 0, SWT.LEFT);
data.right = new FormAttachment(categories, -5);
data.bottom = new FormAttachment(ownerInfo, -5);
dogPhoto.setLayoutData(data);
dogPhoto.addPaintListener(new PaintListener() {
    public void paintControl(final PaintEvent event) {
        if (dogImage != null) {
            event.gc.drawImage(dogImage, 0, 0);
        }
    }
});

data = new FormData();
data.top = new FormAttachment(dogPhoto, 0, SWT.TOP);
photo.setLayoutData(data);

data = new FormData();
data.top = new FormAttachment(photo, 5);
data.right = new FormAttachment(dogPhoto, -5);
browse.setLayoutData(data);
browse.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        String fileName = new FileDialog(shell).open();
        if (fileName != null) {
            dogImage = new Image(display, fileName);
        }
    }
});

data = new FormData();
data.left = new FormAttachment(browse, 0, SWT.LEFT);
```

```
data.top = new FormAttachment(browse, 5);
data.right = new FormAttachment(dogPhoto, -5);
delete.setLayoutData(data);
delete.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        if (dogImage != null) {
            dogImage.dispose();
            dogImage = null;
            dogPhoto.redraw();
        }
    }
});

data = new FormData(90, 140);
data.top = new FormAttachment(dogPhoto, 0, SWT.TOP);
data.right = new FormAttachment(100, 0);
data.bottom = new FormAttachment(enter, -5);
categories.setLayoutData(data);

data = new FormData();
data.bottom = new FormAttachment(categories, -5);
data.left = new FormAttachment(categories, 0, SWT.CENTER);
cats.setLayoutData(data);

data = new FormData();
data.right = new FormAttachment(100, 0);
data.bottom = new FormAttachment(100, 0);
enter.setLayoutData(data);
enter.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        System.out.println("\nDog Name: " + dogNameText.getText());
        System.out.println("Dog Breed: " + dogBreedCombo.getText());
        System.out.println("Owner Name: " + nameText.getText());
        System.out.println("Owner Phone: " + phoneText.getText());
        System.out.println("Categories:");
        String cats[] = categories.getSelection();
        for (int i = 0; i < cats.length; i++) {
            System.out.println("\t" + cats[i]);
        }
    }
});

data = new FormData();
data.bottom = new FormAttachment(enter, -5);
data.left = new FormAttachment(0, 0);
data.right = new FormAttachment(categories, -5);
ownerInfo.setLayoutData(data);

Label name = new Label(ownerInfo, SWT.NULL);
name.setText("Name:");
Label phone = new Label(ownerInfo, SWT.PUSH);
```

```
        phone.setText("Phone:");
        nameText = new Text(ownerInfo, SWT.SINGLE | SWT.BORDER);
        phoneText = new Text(ownerInfo, SWT.SINGLE | SWT.BORDER);

        data = new FormData();
        data.top = new FormAttachment(nameText, 0, SWT.CENTER);
        name.setLayoutData(data);

        data = new FormData();
        data.top = new FormAttachment(phoneText, 0, SWT.CENTER);
        phone.setLayoutData(data);

        data = new FormData();
        data.left = new FormAttachment(phone, 5);
        data.right = new FormAttachment(100, 0);
        nameText.setLayoutData(data);

        data = new FormData();
        data.left = new FormAttachment(nameText, 0, SWT.LEFT);
        data.right = new FormAttachment(100, 0);
        data.top = new FormAttachment(55, 0);
        phoneText.setLayoutData(data);

        shell.pack();
        shell.open();

        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
}
```

上例运行效果和 GridLayout 一样，用户可以通过运行这两个不同的程序，比较这两种方式的布局效果。

7.6 创建自己的布局

在 SWT 中，用户可以通过 setLayout 设置组件的布局信息。布局对象会根据父组件的大小和子组件的布局信息计算出每个子组件的位置和大小，使整个布局空间符合用户的需求。下面将介绍如何创建自己的布局类，实现用户自定义的布局。

7.6.1 Layout 类

在 SWT 中，所有的布局类都继承于 Layout 抽象类。Layout 有两个抽象方法。

（1）computeSize (Composite composite, int wHint, int hHint, boolean flushCache)

（2）layout (Composite composite, boolean flushCache)

computeSize 方法负责计算组件所有子组件所占的高度和宽度，并返回一个 Point 类型的变量（width,height）。layout 方法负责计算子组件的大小和位置，并按计算出来的位置排列子组件。

7.6.2 创建自己的布局类

如果用户希望组件按自己的方式进行布局，可以创建自己的布局类，实现自己的布局。要实现自己的布局，用户要继承 Layout 类，并实现 layout 方法和 computeSize 方法。下面将实现一个简单的按列进行布局的布局类，在此布局中，所有的子组件将按一列显示，并且子组件的宽度相等，代码如例程 7-5 所示。

例程 7-5 ColumnLayout.java

```
package com.free.layout;

import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Layout;

public class ColumnLayout extends Layout {

    public static final int MARGIN = 4;
    public static final int SPACING = 2;

    Point [] sizes;
    int maxWidth, totalHeight;

  protected Point computeSize(Composite composite, int wHint, int hHint,
     boolean flushCache) {

      Control children[] = composite.getChildren();
    if (flushCache || sizes == null || sizes.length != children.length) {
       initialize(children);
    }

    int width = wHint, height = hHint;
    if (wHint == SWT.DEFAULT) width = maxWidth;
    if (hHint == SWT.DEFAULT) height = totalHeight;
    return new Point(width + 2 * MARGIN, height + 2 * MARGIN);
  }

  protected void layout(Composite composite, boolean flushCache) {

    Control children[] = composite.getChildren();
    if (flushCache || sizes == null || sizes.length != children.length) {
```

```
            initialize(children);
        }

        Rectangle rect = composite.getClientArea();
        int x = MARGIN, y = MARGIN;
        //计算最大宽度
        int width = Math.max(rect.width - 2 * MARGIN, maxWidth);
        for (int i = 0; i < children.length; i++) {

            int height = sizes[i].y;
            //设置了组件的位置
            children[i].setBounds(x, y, width, height);
            //计算当前组件的 y 轴的坐标
            y += height + SPACING;
        }
    }

    void initialize(Control children[]) {

        maxWidth = 0;
        totalHeight = 0;
        sizes = new Point [children.length];

        for (int i = 0; i < children.length; i++) {
            //计算了组件的大小
            sizes[i] = children[i].computeSize(SWT.DEFAULT, SWT.DEFAULT, true);
            maxWidth = Math.max(maxWidth, sizes[i].x);
            totalHeight += sizes[i].y;
        }
        totalHeight += (children.length - 1) * SPACING;
    }
    }
```

在 ColumnLayout 类中，通过 layout 方法对子组件重新计算位置，并设置子组件的位置。为了验证 ColumnLayout 类，下面通过 ColumnLayoutTest 类测试 ColumnLayout 类的效果，代码如例程 7-6 所示。

例程 7-6　ColumnLayoutTest.java

```
package com.free.layout;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class ColumnLayoutTest {
```

```
static Shell shell;
static Button button3;

public static void main(String[] args) {

    Display display = new Display();
    shell = new Shell(display);
    shell.setLayout(new ColumnLayout());
    new Button(shell, SWT.PUSH).setText("B1");
    new Button(shell, SWT.PUSH).setText("Very Wide Button 2");
    (button3 = new Button(shell, SWT.PUSH)).setText("Button 3");
    new Text(shell, SWT.NONE).setText("text");
    Button grow = new Button(shell, SWT.PUSH);

    grow.setText("Grow Button 3");
    // 添加选择组件监听器
    grow.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {

        button3.setText("Extreemely Wide Button 3");
        //组件大小改变后通知父组件进行重新布局
        shell.layout();
        shell.pack();
      }
    });

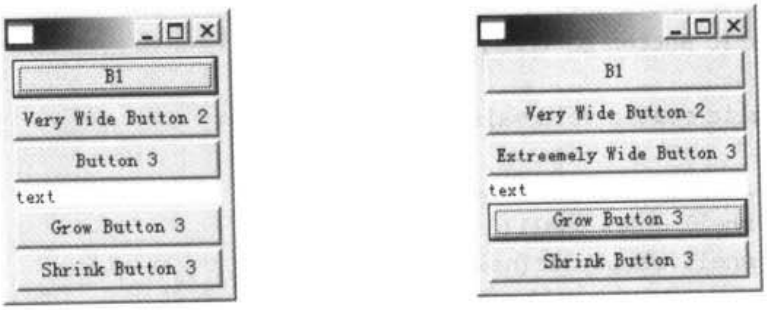
    Button shrink = new Button(shell, SWT.PUSH);
    shrink.setText("Shrink Button 3");
    shrink.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {

        button3.setText("Button 3");
        //组件大小改变后通知父组件进行重新布局
        shell.layout();
        shell.pack();
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch()) display.sleep();
    }
  }
}
```

当选择“Grow Button 3”组件后，layout方法会根据子组件的最大宽度调整所有子组件的宽度，程序运行效果如图7-6所示。



原始大小　　宽度改变后

图 7-6　自己定义布局

7.7　小　　结

本章通过实例介绍了几种常用的布局方式，读者可以通过这几种布局方式实现 SWT 中大多数的布局需求。如果有某些比较特殊的要求，读者可以尝试修改布局类，以适应相关的布局。在本章，读者应该掌握如何设置组件相应的布局信息，掌握如何使用几种方式进行布局，特别是 GridLayout 布局方式。在有特殊需要的时候要能够修改布局类以适应自己的要求。

第 8 章　SWT 容器及其他组件

SWT 提供了操作系统中常用组件的封装，本章主要介绍容器和其他的一些组件。本章内容包括如下几个部分。

- SWT 容器
- 菜单和工具栏
- 其他组件

8.1　SWT 容器

容器是 SWT 中常用的组件，它能够作为其他组件的父组件，通过设置布局父组件的布局信息对子组件进行布局。SWT 中常用的组件有面板（Composite）、分组框（Group）、分页框（Tab）和分割框（SashForm）。

8.1.1　Composite 组件

Composite 是 SWT 中常用的组件，它的作用相当于一个容器面板，用户可以创建一个面板对象，通过面板对象来添加其他的子组件。

在 GUI 的开发中，为了布局的方便，常常用 Composite 来添加子组件，在 Composite 上设置相应的布局，从而使局部的布局效果不一样。一般来说面板可以设置成 SWT.NONE 的风格，另外还可以给 Composite 设置成带边框的风格。创建一个面板很简单，代码如例程 8-1 所示。

例程 8-1　ComposizeExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Shell;

public class ComposizeExample {

    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
```

```
        shell.setLayout(new FillLayout());

        //创建面板
        Composite composite1 = new Composite(shell, SWT.BORDER);
        composite1.setLayout(new RowLayout(SWT.VERTICAL));
        new Button(composite1, SWT.RADIO).setText("John");
        new Button(composite1, SWT.RADIO).setText("Paul");
        new Button(composite1, SWT.RADIO).setText("George");
      new Button(composite1, SWT.RADIO).setText("Ringo");

      Composite composite2 = new Composite(shell, SWT.NONE);
      composite2.setLayout(new RowLayout(SWT.VERTICAL));
       new Button(composite2, SWT.CHECK).setText("Barry");
       new Button(composite2, SWT.CHECK).setText("Robin");
       new Button(composite2, SWT.CHECK).setText("Maurice");

       shell.setSize(200, 200);
       shell.open();
       while (!shell.isDisposed()) {
         if (!display.readAndDispatch()) {
           display.sleep();
         }
       }
       display.dispose();

    }

}
```

以上程序创建了两个面板，一个设置成 SWT.BORDER 风格（带边框），另一个设置成 SWT.NONE 风格（未添加风格），程序运行效果如图 8-1 所示。

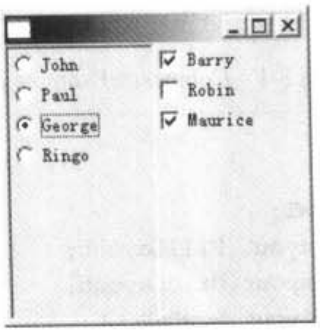


图 8-1　Composite 组件

8.1.2　Group 组件

在 SWT 中，Group 组件代表分组框，用户可以通过分组框把内容上相关的子组件组合在一起，例如某一类人、某一种水果等。

用户可以通过 Group 建立一个特定的分组，每一个分组有一个名称，可以通过 Group 的 setText 设置分组的显示名称，代码如例程 8-2 所示。

例程 8-2　GroupExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Shell;

public class GroupExample {
      public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setLayout(new FillLayout());

        //添加分组框
        Group group1 = new Group(shell, SWT.SHADOW_IN);
        //设置分组框的显示标题
        group1.setText("Who's your favorite?");
        group1.setLayout(new RowLayout(SWT.VERTICAL));
        new Button(group1, SWT.RADIO).setText("John");
        new Button(group1, SWT.RADIO).setText("Paul");
        new Button(group1, SWT.RADIO).setText("George");
       new Button(group1, SWT.RADIO).setText("Ringo");

        Group group2 = new Group(shell, SWT.NO_RADIO_GROUP);
        group2.setText("Who's your favorite?");
        group2.setLayout(new RowLayout(SWT.VERTICAL));
        new Button(group2, SWT.CHECK).setText("Barry");
        new Button(group2, SWT.CHECK).setText("Robin");
        new Button(group2, SWT.CHECK).setText("Maurice");

        shell.setSize(300, 200);
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
            display.sleep();
          }
        }
        display.dispose();
      }
    }
```

以上程序创建了两个分组框，每个分组框都设定了相关显示标题，程序运行效果如图 8-2 所示。

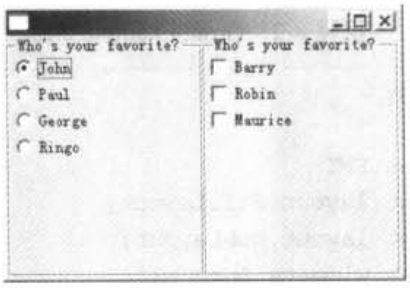


图 8-2 Group 组件

8.1.3 TabFolder 和 TabItem 组件

在很多应用程序中，由于信息量比较大，一页可能显示不下所有的信息，这就要求多页组件的支持。

在 SWT 中，通过 TabFolder 组织分页框，TabItem 为每一个分页。创建一个分页框的步骤如下。

（1）在容器中创建一个 TabFolder 对象，例如“TabFolder tabFolder = new TabFolder(shell, SWT.BORDER);”。

（2）在 TabFolder 中创建一个 TabItem 对象，例如“TabItem tabItem = new TabItem (tabFolder, SWT.NULL);”。

（3）设置 TabItem 的显示标签，例如“tabItem.setText("Very good");”。

（4）设置每一分页的 Control 对象（即在此页的顶层组件，一般是容器），例如“tabItem.setControl(composite);”。

为了更好地掌握分页框，下面通过一个实例演示如何创建分页框，代码如例程 8-3 所示。

例程 8-3 TabFolderExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.TabFolder;
import org.eclipse.swt.widgets.TabItem;
import org.eclipse.swt.widgets.Text;

public class TabFolderExample {
      public static void main(String[] args) {
        Display display = new Display();
        final Shell shell = new Shell(display);
        shell.setText("Tab Folder Example");
        shell.setSize(450, 250);

        //创建分页框
        final TabFolder tabFolder = new TabFolder(shell, SWT.BORDER);
```

```
        for (int loopIndex = 0; loopIndex < 3; loopIndex++) {
          //创建分页项
          TabItem tabItem = new TabItem(tabFolder, SWT.NULL);
          //设置分页项的标题
          tabItem.setText("Tab " + loopIndex);

          Text text = new Text(tabFolder, SWT.BORDER);
          text.setText("This is page " + loopIndex);
          //设置 Control 对象
          tabItem.setControl(text);
        }

        TabItem tabItem = new TabItem(tabFolder, SWT.NULL);
        tabItem.setText("Tab " + 3);
        Composite composite = new Composite(tabFolder, SWT.BORDER);
        Text text = new Text(composite, SWT.BORDER);
        text.setText("This is page " + 3);
        text.setBounds(10, 10, 100, 20);
        //设置 Control 对象
        tabItem.setControl(composite);


        tabFolder.setSize(400, 200);

        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch())
            display.sleep();
        }
        display.dispose();
      }
    }
```

以上程序创建了 4 个分页，其中第 4 个分页的 Control 对象为一个容器，程序运行效果如图 8-3 所示。

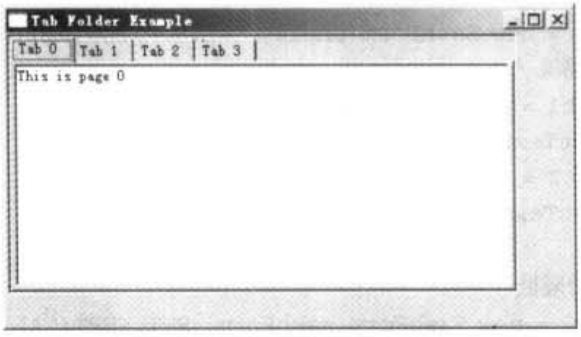


图 8-3　TabFolder\TabItem 组件

8.1.4　SashForm 组件

SashForm 是 SWT 中的分隔框组件，在容器中，有时需要调整容器中元素的大小，这就要

用到 SashForm。在 SWT 中添加分隔框的步骤如下。

（1）在容器中创建 SashForm 对象，例如“sashForm = new SashForm(shell, SWT.HORIZONTAL);”。

（2）在 SashForm 对象中添加分隔项（即分隔框的子组件），可以是 SashForm 或 Control 的子类，例如“Text text1 = new Text(sashForm, SWT.CENTER);”。

（3）设置 SashForm 中分隔项显示比例的权重（可选），例如“sashForm.setWeights(new int[]{1, 2, 3});”，表示 3 个分隔项之间的显示比例为“1:2:3”。

为了更好地掌握分隔框，下面通过一个实例演示如何创建分隔框，代码如例程 8-4 所示。

例程 8-4 SashFormExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.custom.SashForm;
import org.eclipse.swt.events.ControlEvent;
import org.eclipse.swt.events.ControlListener;
import org.eclipse.swt.events.MouseEvent;
import org.eclipse.swt.events.MouseListener;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class SashFormExample {
      Display display = new Display();
      Shell shell = new Shell(display);
      SashForm sashForm;
      SashForm sashForm2;

      public SashFormExample() {
        shell.setLayout(new FillLayout());
        //添加分隔框
        sashForm = new SashForm(shell, SWT.HORIZONTAL);
        //添加分隔项
        Text text1 = new Text(sashForm, SWT.CENTER);
        text1.setText("Text in pane #1");
        Text text2 = new Text(sashForm, SWT.CENTER);
        text2.setText("Text in pane #2");

        //添加子分隔框
        sashForm2 = new SashForm(sashForm, SWT.VERTICAL);
        final Label labelA = new Label(sashForm2, SWT.BORDER | SWT.CENTER);
        labelA.setText("Label in pane A");
        final Label labelB = new Label(sashForm2, SWT.BORDER |SWT.CENTER);
        labelB.setText("Label in pane B");

        //添加 ControlListener 监听器
```

```
text1.addControlListener(new ControlListener() {
  public void controlMoved(ControlEvent e) {
  }

  public void controlResized(ControlEvent e) {
    System.out.println("Resized");
  }
});
//设置分隔框中组件显示比例的权重
sashForm.setWeights(new int[]{1, 2, 3});
labelA.addMouseListener(new MouseListener() {
  public void mouseDoubleClick(MouseEvent e) {
    if(sashForm2.getMaximizedControl() == labelA)
      sashForm2.setMaximizedControl(null);
    else
      sashForm2.setMaximizedControl(labelA);
  }

  public void mouseDown(MouseEvent e) {
  }

  public void mouseUp(MouseEvent e) {
  }
});
shell.setSize(450, 200);
shell.open();
while (!shell.isDisposed()) {
  if (!display.readAndDispatch()) {
    // If no more entries in event queue
    display.sleep();
  }
}

display.dispose();
}

public static void main(String[] args) {
  new SashFormExample();
}
}
```

以上程序中为嵌套的分隔框组件，第一层分隔项的显示比例为“1:2:3”，程序运行效果如图 8-4 所示。

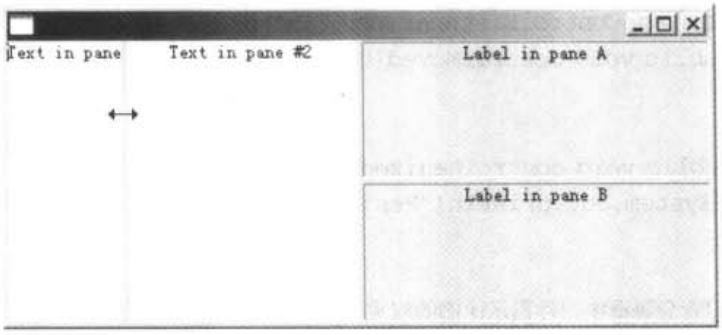


图 8-4 SashForm 组件

8.2 菜单和工具栏

菜单和工具栏是窗口系统中常用的表现方式，本节将介绍 SWT 中菜单和工具栏的使用方式，具体如下。

8.2.1 Menu 和 MenuItem 组件

在程序中，菜单是提供信息比较好的方式，SWT 中通过 Menu 和 MenuItem 实现菜单和菜单项，在程序中添加菜单的步骤如下。

（1）创建 Menu 对象，并指定创建的风格，例如“menuBar = new Menu(shell, SWT.BAR);”。

（2）创建 MenuItem 对象，并指定创建风格，例如“fileEnglishItem = new MenuItem (fileMenu, SWT.RADIO);”。

（3）设置 Menu 和 MenuItem 的关联（Menu 中还可以有子 Menu），例如“fileMenuHeader.setMenu(fileMenu);”。

（4）添加 MenuItem 的事件监听器，例如“fileEnglishItem.addSelectionListener(new RadioItemListener());”。

为了更好地掌握菜单，下面通过一个实例演示如何创建菜单，代码如例程 8-5 所示。

例程 8-5 MenuExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.widgets.MenuItem;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class MenuExample {
```

```
Display display;

Shell shell;

Menu menuBar, fileMenu, editMenu;

MenuItem fileMenuHeader, editMenuHeader;

MenuItem fileExitItem, fileSaveItem, fileEnglishItem, fileGermanItem,
    editCopyItem;

Text text;

public MenuExample() {
  display = new Display();
  shell = new Shell(display);
  shell.setText("Menu Example");
  shell.setSize(300, 200);

  text = new Text(shell, SWT.BORDER);
  text.setBounds(80, 50, 150, 25);

  //添加主菜单项
  menuBar = new Menu(shell, SWT.BAR);
  //添加一级子菜单
  fileMenuHeader = new MenuItem(menuBar, SWT.CASCADE);
  fileMenuHeader.setText("&File");

 //添加 一级子菜单的菜单项
  fileMenu = new Menu(shell, SWT.DROP_DOWN);
  fileMenuHeader.setMenu(fileMenu);

  fileSaveItem = new MenuItem(fileMenu, SWT.PUSH);
  fileSaveItem.setText("&Save");

  fileEnglishItem = new MenuItem(fileMenu, SWT.RADIO);
  fileEnglishItem.setText("English");

  fileGermanItem = new MenuItem(fileMenu, SWT.RADIO);
  fileGermanItem.setText("German");

  fileExitItem = new MenuItem(fileMenu, SWT.PUSH);
  fileExitItem.setText("E&xit");

  editMenuHeader = new MenuItem(menuBar, SWT.CASCADE);
  editMenuHeader.setText("&Edit");

  editMenu = new Menu(shell, SWT.DROP_DOWN);
  editMenuHeader.setMenu(editMenu);

  editCopyItem = new MenuItem(editMenu, SWT.PUSH);
```

```
        editCopyItem.setText("&Copy");

        //添加菜单项的事件监听器
        fileExitItem.addSelectionListener(new MenuItemListener());
        fileSaveItem.addSelectionListener(new MenuItemListener());
        editCopyItem.addSelectionListener(new MenuItemListener());

        fileEnglishItem.addSelectionListener(new RadioItemListener());
        fileGermanItem.addSelectionListener(new RadioItemListener());

        shell.setMenuBar(menuBar);
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch())
            display.sleep();
        }
        display.dispose();
      }

      class MenuItemListener extends SelectionAdapter {
        public void widgetSelected(SelectionEvent event) {
              text.setText("You selected " + ((MenuItem) event.widget).getText());
              if (((MenuItem) event.widget).getText().equals("E&xit")) {
                  shell.close();
              }
          }
      }

      class RadioItemListener extends SelectionAdapter {
        public void widgetSelected(SelectionEvent event) {
          MenuItem item = (MenuItem) event.widget;
          text.setText(item.getText() + " is on.");
        }
      }

      public static void main(String[] args) {
        MenuExample menuExample = new MenuExample();
      }
    }
```

以上程序中添加了主菜单，并在主菜单中添加了两个子菜单项，子菜单项添加了相应的事件响应机制，程序运行效果如图 8-5 所示。

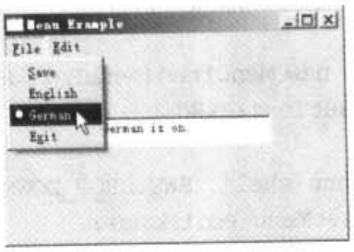


图 8-5　Menu\MenuItem 组件

8.2.2　ToolBar 和 ToolItem 组件

ToolBar 是 SWT 中的工具栏组件，ToolItem 是工具栏中的工具项（一般表现为按钮或分隔符，也可以是其他组件），在程序中添加工具栏的步骤如下。

（1）创建 ToolBar 对象，并指定创建的风格，例如“toolBar = new ToolBar(shell, SWT.FLAT | SWT.WRAP | SWT.RIGHT);”。

（2）创建 ToolItem 对象，并指定创建风格，例如“ToolItem itemPush = new ToolItem (toolBar, SWT.PUSH);”。

（3）设置 ToolItem 的图标和相关属性，例如“itemPush.setImage(icon);”。

（4）添加 ToolItem 的事件监听器，例如“itemPush.addListener(SWT.Selection,selectionListener);”。

为了更好地掌握工具栏组件，下面通过一个实例演示如何创建工具栏组件，代码如例程 8-6 所示。

例程 8-6　ToolBarExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.graphics.Rectangle;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import org.eclipse.swt.widgets.Listener;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.widgets.MenuItem;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;
import org.eclipse.swt.widgets.ToolBar;
import org.eclipse.swt.widgets.ToolItem;

public class ToolBarExample {
    Display display = new Display();

    Shell shell = new Shell(display);

    ToolBar toolBar;

    public ToolBarExample() {
        //添加工具栏
        toolBar = new ToolBar(shell, SWT.FLAT | SWT.WRAP | SWT.RIGHT);
        //添加工具项
        ToolItem itemPush = new ToolItem(toolBar, SWT.PUSH);
        itemPush.setText("PUSH item");
        //设置工具项的显示图标
        Image icon = new Image(shell.getDisplay(), "icons/new.gif");
        itemPush.setImage(icon);
        ToolItem itemCheck = new ToolItem(toolBar, SWT.CHECK);
```

```
itemCheck.setText("CHECK item");
ToolItem itemRadio1 = new ToolItem(toolBar, SWT.RADIO);
itemRadio1.setText("RADIO item 1");
ToolItem itemRadio2 = new ToolItem(toolBar, SWT.RADIO);
itemRadio2.setText("RADIO item 2");
ToolItem itemSeparator = new ToolItem(toolBar, SWT.SEPARATOR);

final ToolItem itemDropDown = new ToolItem(toolBar, SWT.DROP_DOWN);
itemDropDown.setText("DROP_DOWN item");
itemDropDown.setToolTipText("Click here to see a drop down menu ...");
final Menu menu = new Menu(shell, SWT.POP_UP);
new MenuItem(menu, SWT.PUSH).setText("Menu item 1");
new MenuItem(menu, SWT.PUSH).setText("Menu item 2");
new MenuItem(menu, SWT.SEPARATOR);
new MenuItem(menu, SWT.PUSH).setText("Menu item 3");
//设置工具项的事件监听器
itemDropDown.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event event) {
        if (event.detail == SWT.ARROW) {
            Rectangle bounds = itemDropDown.getBounds();
            Point point = toolBar.toDisplay(bounds.x, bounds.y
                    + bounds.height);
            //设置菜单的显示位置
            menu.setLocation(point);
            menu.setVisible(true);
        }
    }
});
//设置工具项的事件监听器
Listener selectionListener = new Listener() {
    public void handleEvent(Event event) {
        ToolItem item = (ToolItem) event.widget;
        System.out.println(item.getText() + " is selected");
        if ((item.getStyle() & SWT.RADIO) != 0
                || (item.getStyle() & SWT.CHECK) != 0)
            System.out.println("Selection status: "
                    + item.getSelection());
    }
};
itemPush.addListener(SWT.Selection, selectionListener);
itemCheck.addListener(SWT.Selection, selectionListener);
itemRadio1.addListener(SWT.Selection, selectionListener);
itemRadio2.addListener(SWT.Selection, selectionListener);
itemDropDown.addListener(SWT.Selection, selectionListener);

toolBar.pack();
shell.addListener(SWT.Resize, new Listener() {
    public void handleEvent(Event event) {
        Rectangle clientArea = shell.getClientArea();
        toolBar.setSize(toolBar.computeSize(clientArea.width,
                SWT.DEFAULT));
```

```
            }
        });
        shell.setSize(400, 100);
        shell.open();

        while (!shell.isDisposed()) {
            if (!display.readAndDispatch()) {
                display.sleep();
            }
        }

        display.dispose();
    }

    public static void main(String[] args) {
        new ToolBarExample();
    }
}
```

程序添加了工具栏，并在工具栏中添加了相应的工具项，工具项中添加了相应的事件响应机制，程序运行效果如图 8-6 所示。

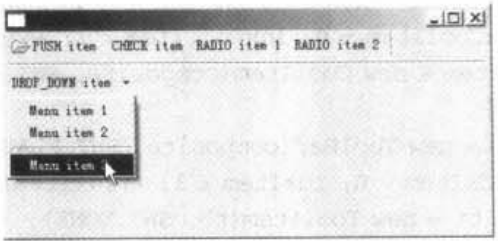


图 8-6　工具栏组件

8.2.3　CoolBar 和 CoolItem 组件

CoolBar 是另外一种形式的工具栏，它能够调整 CoolItem 的位置，每一个 CoolItem 可以设定相关的组件（也可以是另一个工具栏），创建 CoolBar 的步骤如下。

（1）创建 CoolBar 对象，并指定创建的风格，例如“CoolBar composite = new CoolBar (parent, SWT.NONE);”。

（2）创建 CoolItem 对象，并指定创建风格，例如“CoolItem item = new CoolItem(composite, SWT.NONE);”。

（3）设置 CoolItem 的 Control 对象，例如“item.setControl(tb);”。

CoolBar 相当于一个面板容器，CoolItem 是容器中的每一项，CoolItem 能设置相应的组件作为此项的子容器（也可以是其他组件）。为了更好地掌握 CoolBar 组件，下面通过一个实例演示如何创建 CoolBar 组件，代码如例程 8-7 所示。

例程 8-7　CoolBarExample.java

```
package com.free.show;
```

```
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.CoolBar;
import org.eclipse.swt.widgets.CoolItem;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.ToolBar;
import org.eclipse.swt.widgets.ToolItem;

public class CoolBarExample extends ApplicationWindow {
      public CoolBarExample() {
        super(null);
      }

      protected Control createContents(Composite parent) {
        getShell().setText("CoolBar Test");

        String asCoolItemSection[] = { "File", "Formatting", "Search" };
        //添加CoolBar
        CoolBar composite = new CoolBar(parent, SWT.NONE);
        for (int idxCoolItem = 0; idxCoolItem < 3; ++idxCoolItem) {
          CoolItem item = new CoolItem(composite, SWT.NONE);
          //添加子组件
          ToolBar tb = new ToolBar(composite, SWT.FLAT);
          for (int idxItem = 0; idxItem < 3; ++idxItem) {
            ToolItem ti = new ToolItem(tb, SWT.NONE);
            ti
                .setText(asCoolItemSection[idxCoolItem] + " Item #"
                    + idxItem);
          }
          Point p = tb.computeSize(SWT.DEFAULT, SWT.DEFAULT);
          tb.setSize(p);
          Point p2 = item.computeSize(p.x, p.y);

          //设置为一个CoolItem的控制类
          item.setControl(tb);
          item.setSize(p2);
        }
        return composite;
      }

      public static void main(String[] args) {
        CoolBarExample app = new CoolBarExample();
        app.setBlockOnOpen(true);
        app.open();
        Display.getCurrent().dispose();
      }
    }
```

以上代码演示了如何创建 CoolBar。CoolBar 中每一个 CoolItem 可以根据用户的需要调整位置，程序运行效果如图 8-7 所示。

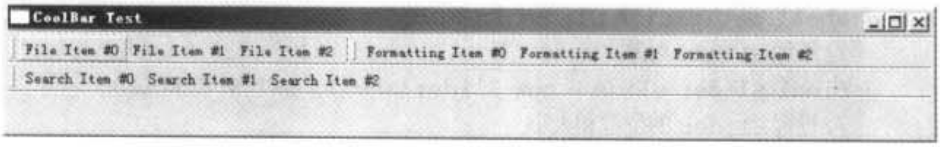


图 8-7　CoolBar 组件

8.3　其 他 组 件

SWT 组件是非常丰富的，本书无法一一介绍，下面就几个常用 SWT 组件介绍如下。

8.3.1　Slider 组件

为了方便用户输入数据，SWT 中提供了 Slider 组件，用户可通过 Slider 设置数据的增量值，用来控制其他组件，也可以作为滚动条控制其他组件中的数据显示。添加 Slider 组件的步骤如下。

（1）创建 Slider 对象，并指定创建的风格，例如“Slider slide = new Slider(shell, SWT.HORIZONTAL);”。

（2）设置 Slider 的最大值和最小值，例如“slide.setMaximum(100);”。

（3）设置 Slider 增量的增加或递减值，例如“slide.setIncrement(1);”。

（4）添加 Slider 的事件监听器，例如“slide.addSelectionListener(selectionListener);”。

为了更好地掌握 Slider 组件，下面通过一个实例演示如何创建 Slider 组件，代码如例程 8-8 所示。

例程 8-8　SliderExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Slider;
import org.eclipse.swt.widgets.Text;

public class SliderExample {

    Display dispaly;

    Shell shell;

    SliderExample() {
```

```
        dispaly = new Display();
        shell = new Shell(dispaly);
        shell.setSize(300, 250);
        shell.setText("A Slider Example");
        //添加Slider对象
        final Slider slide = new Slider(shell, SWT.V_SCROLL);
        //设置Slider的位置和大小
        slide.setBounds(170, 25, 25, 20);
        //设置Slider的最小值
        slide.setMinimum(0);
        //设置Slider的最大值
        slide.setMaximum(100);
        //设置Slider单击左右箭头的增加或递减值
        slide.setIncrement(1);

        final Text t = new Text(shell, SWT.BORDER);
        t.setBounds(115, 25, 55, 20);
        t.setText("0");
        t.setFocus();

        //添加Slider的事件监听器
        slide.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent e) {
                t.setText(new Integer(slide.getSelection()).toString());
            }
        });

        shell.open();
        while (!shell.isDisposed()) {
            if (!dispaly.readAndDispatch())
                dispaly.sleep();
        }
        dispaly.dispose();
    }

    public static void main(String[] args) {
        new SliderExample();
    }

}
```

以上代码添加了一个 Text 组件和一个 Slider 组件。Slider 组件设置了增量值为 1，另外 Slider 组件添加了选择事件，当选择了 Slider 组件后，Slider 将为 Text 组件赋值。程序运行效果如图 8-8 所示。

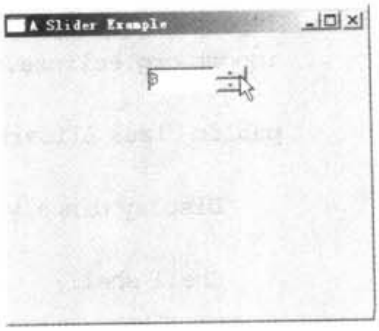


图 8-8 Slider 组件

8.3.2 Scale 组件

Scale 和 Slider 类似，在 SWT 中都表示一种尺度，但两

者的表现形式不一样，Scale 更像一个刻度，而 Slider 则是提供一个滚动条。添加 Scale 组件的步骤如下。

（1）创建 Scale 对象，并指定创建的风格，例如“Scale scale = new Scale(shell, SWT.VERTICAL);”。

（2）设置 Scale 的最大值和最小值，例如“scale.setMaximum(20);”。

（3）设置 Scale 增量的增加或递减值，例如“scale.setPageIncrement(5);”。

（4）添加 Scale 的事件监听器，例如“scale.addSelectionListener(selectionListener);”。

为了更好地掌握 Scale 组件，下面通过一个实例演示如何创建 Scale 组件，代码如例程 8-9 所示。

例程 8-9　ScaleExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Listener;
import org.eclipse.swt.widgets.Scale;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class ScaleExample {
      Display display = new Display();
      Shell shell = new Shell(display);

      Scale scale;
      Text value;
      public ScaleExample() {
        shell.setLayout(new GridLayout(1, true));
        Label label = new Label(shell, SWT.NULL);
        label.setText("Volume:");
        //添加Scale组件
        scale = new Scale(shell, SWT.VERTICAL);
        scale.setBounds(0, 0, 40, 200);
        //设置Scale的最大值
        scale.setMaximum(20);
        //设置Scale的最小值
        scale.setMinimum(0);
        //设置Scale的增量值
        scale.setPageIncrement(5);
        //添加Scale的事件监听器
        scale.addListener(SWT.Selection, new Listener() {
          public void handleEvent(Event event) {
            int perspectiveValue = scale.getMaximum() - scale.getSelection() +
            scale.getMinimum();
```

```
            value.setText("Vol: " + perspectiveValue);
          }
        });
        value = new Text(shell, SWT.BORDER | SWT.SINGLE);

        value.setEditable(false);
        scale.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_CENTER));
        value.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_CENTER));

        shell.pack();
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch()) {
            display.sleep();
          }
        }

        display.dispose();
      }

      private void init() {

      }

      public static void main(String[] args) {
        new ScaleExample();
      }
    }
```

上例中，通过事件监听器监听当前选择的刻度，并用 Text 组件显示出来，程序运行效果如图 8-9 所示。

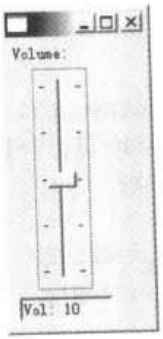


图 8-9　Scale 组件

8.3.3　ProgressBar 组件

ProgressBar 是 SWT 中的进度条组件。进度条提供了比较长时间操作的进度信息。添加 ProgressBar 组件的步骤如下。

（1）创建 ProgressBar 对象，并指定创建的风格，例如“ProgressBar pb1 = new ProgressBar (shell, SWT.HORIZONTAL | SWT.SMOOTH);”。

（2）设置 ProgressBar 的最大值和最小值，例如“pb1.setMaximum(30);”。

（3）在长时间的任务中设置当前进度条的进度，例如“progressBar.setSelection (progressBar.getSelection() + 1);”。

进度条能反映当前的工作进度，为了配合处理长时间的任务，进度条经常配合线程使用，以免产生阻塞影响界面的操作。为了更好地掌握 ProgressBar 组件，下面通过一个实例演示如何创建 ProgressBar 组件，代码如例程 8-10 所示。

例程 8-10 ProgressBarExample.java

```
package com.free.show;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.ProgressBar;
import org.eclipse.swt.widgets.Shell;

public class ProgressBarExample {
    public static void main(String[] args) {
      Display display = new Display();
      Shell shell = new Shell(display);
      shell.setLayout(new GridLayout());
      //添加平滑的进度条
      ProgressBar pb1 = new ProgressBar(shell, SWT.HORIZONTAL | SWT.SMOOTH);
      pb1.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
      //设置进度条的最小值
      pb1.setMinimum(0);
      //设置进度条的最大值
      pb1.setMaximum(30);
      //添加自动递增的进度条
      ProgressBar pb2 = new ProgressBar(shell, SWT.HORIZONTAL |
        SWT.INDETERMINATE);
      pb2.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
      //添加线程，在线程中处理长时间的任务，并最终反映在平滑进度条上
      new LongRunningOperation(display, pb1).start();
      shell.open();
      while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) {
          display.sleep();
        }
      }
    }
    class LongRunningOperation extends Thread {
      private Display display;
      private ProgressBar progressBar;

      public LongRunningOperation(Display display, ProgressBar progressBar) {
        this.display = display;
```

```
        this.progressBar = progressBar;
    }
    public void run() {
        //模仿长时间的任务
        for (int i = 0; i < 30; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {

            }
            display.asyncExec(new Runnable() {
                public void run() {
                    if (progressBar.isDisposed()) return;
                    //进度条递增
                    progressBar.setSelection(progressBar.getSelection() + 1);
                }
            });
        }
    }
}
```

以上代码添加了两个进度条，一个进度条为自动显示增加进度的信息（SWT.INDETERMINAT风格），另外一个进度条通过线程处理长时间的任务，并设定进度条的信息。程序运行效果如图 8-10 所示。

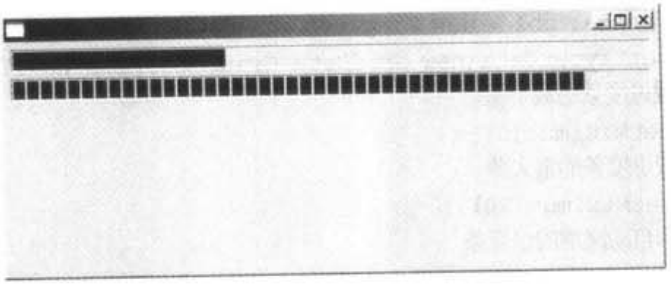


图 8-10　ProgressBar 组件

8.4　小　　结

本章介绍了容器和一些其他组件，这些组件是比较常用的，不过读者没必要全部记住，只需要知道在什么时候使用什么样的组件即可。

第 9 章　Viewer 组件

在 SWT 界面的开发中，经常会用到表、树和列表等相关组件，JFace 提供了几个功能强大的组件。

本章将通过实例介绍 JFace 的如下几个组件。

- TreeViewer
- TableViewer
- ListViewer

9.1　TreeViewer

JFace 中提供了 TreeViewer 组件，用 TreeViewer 来表示一棵树。TreeViewer 功能比较强，它能定义节点、节点的显示标签、响应事件和显示图标等信息。

9.1.1　TreeViewer 构建步骤

TreeViewer 中主要通过内容提供器（ITreeContentProvider）和标签提供器（ILabelProvider）组织树节点的内容和显示的信息。TreeViewer 的构建步骤如下。

（1）创建 TreeViewer 对象，例如“TreeViewer tv = new TreeViewer(composite);”。

（2）设定内容管理器，例如“tv.setContentProvider(new FileTreeContentProvider());”。

（3）设定标签提供器，例如“tv.setLabelProvider(new FileTreeLabelProvider());”。

（4）设定 TreeViewer 的输入数据，例如“tv.setInput("root");”（用户可以通过输入数据构建树）。

9.1.2　内容提供器（ITreeContentProvider）

TreeViewer 的内容提供器（ITreeContentProvider）构建树中比较复杂的部分，它为树的显示提供了内容，内容提供器要实现的方法如下。

（1）getElements

此函数定义为“public Object[] getElements(Object inputElement);”，当程序开始构建树时，首先调用 getElements 返回一个对象的数组，此数组对象表示当前树的根节点，inputElement 参数为 TreeViewer 的输入（setInput 的输入数据）。

（2）hasChildren

此函数定义为“public boolean hasChildren(Object element);”，当 TreeViewer 显示一个节点后，会调用 hasChildren 函数判断当前节点是否有子节点，如果有子节点则显示“+”，element 参数为要判断是否有子节点的节点。

（3）getChildren

此函数定义为“public Object[] getChildren(Object parentElement);”，当用户选择节点打开子节点时，会调用 getChildren 函数返回下一层子节点，parentElement 参数为选择的节点。

（4）getParent

此函数定义为“public Object getParent(Object element);”，可以通过此方法返回 element 的父节点。

（5）inputChanged

此函数定义为“public void inputChanged(Viewer viewer, Object oldInput, Object newInput);”，当输入改变时调用此方法。

（6）dispose

此函数定义为“public void dispose();”，当树销毁时被调用。

其中，getElements、hasChildren 和 getChildren 是常用的方法，用户通过重写这几个方法构建一棵树，过程如下：通过 getElements 方法得到根，再通过 hasChildren 判断根下是否有子节点，如果有子节点，可以通过 getChildren 得到所有的子节点。例程 9-1 为 ITreeContentProvider 接口的一个简单实现。

例程 9-1 FileTreeContentProvider.java

```
class FileTreeContentProvider implements ITreeContentProvider {

    public Object[] getChildren(Object arg0) {
        //返回树的下一级节点
        return ((File) arg0).listFiles();
    }

    public Object getParent(Object arg0) {
        //返回树的上一级节点
        return ((File) arg0).getParentFile();
    }

    public boolean hasChildren(Object arg0) {
        Object[] obj = getChildren(arg0);

        //判断树是否有下一级节点，true 为在节点显示"+"信息
        return obj == null ? false : obj.length > 0;
    }

    public Object[] getElements(Object arg0) {

        //打印出树的输入信息，通常用户可以通过输入信息构建树
        System.out.println(arg0);
        // File.listRoots()作为树的根节点
        return File.listRoots();
    }
```

上例内容提供器通过文件系统获得树的输入内容，从而使用户构造的树能显示磁盘文件的树结构。

9.1.3 标签提供器（ILabelProvider）

在 TreeViewer 中，通过标签提供器（ILabelProvider）来显示节点的相关信息，包括显示内容和图标。

ILabelProvider 主要实现 getImage 和 getText 函数。当 TreeViewer 得到一个节点后会通过 getText 得到此节点的显示文本，通过 getImage 方法得到节点的显示图标，代码如例程 9-2 所示。

例程 9-2 ViewLabelProvider.java

```
class ViewLabelProvider extends ILabelProvider{
…
        public String getText(Object obj) {
            return obj.toString();
        }
        public Image getImage(Object obj) {
            String imageKey = ISharedImages.IMG_OBJ_ELEMENT;
            if (obj instanceof TreeParent)
              imageKey = ISharedImages.IMG_OBJ_FOLDER;
            return PlatformUI.getWorkbench().getSharedImages().getImage(imageKey);
        }
…
    }
```

9.1.4 TreeViewer 实例

在 TreeViewer 实例中，一般还要定义树节点的结构，本例中的 File 对象就能够表示节点的父子关系（在视图扩展点的实现一章，笔者定义了一个树节点，有兴趣的读者可以在此章找到树节点的定义），代码如例程 9-3 所示。

例程 9-3 FileBrowserTreeViewer.java

```
package com.free.tableandtree;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.List;

import org.eclipse.jface.viewers.ILabelProvider;
import org.eclipse.jface.viewers.ILabelProviderListener;
import org.eclipse.jface.viewers.ITreeContentProvider;
import org.eclipse.jface.viewers.LabelProviderChangedEvent;
import org.eclipse.jface.viewers.TreeViewer;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
```

```
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class FileBrowserTreeViewer extends ApplicationWindow {
    public FileBrowserTreeViewer() {
        super(null);
    }

    public void run() {
        setBlockOnOpen(true);

        open();

        Display.getCurrent().dispose();
    }

    protected void configureShell(Shell shell) {
        super.configureShell(shell);

        shell.setText("File Tree");
        shell.setSize(400, 400);
    }

    protected Control createContents(Composite parent) {
        Composite composite = new Composite(parent, SWT.NONE);
        // 父容器分 列显示
        composite.setLayout(new GridLayout(1, false));

        Button preserveCase = new Button(composite, SWT.CHECK);
        preserveCase.setText("&Preserve case");

        final TreeViewer tv = new TreeViewer(composite);
        // 让树组件两端对齐
        tv.getTree().setLayoutData(new GridData(GridData.FILL_BOTH));
        // 添加内容管理器
        tv.setContentProvider(new FileTreeContentProvider());
        // 添加标签管理器
        tv.setLabelProvider(new FileTreeLabelProvider());
        // 设置 treeviewer 的输入
        tv.setInput("root"); // pass a non-null that will be ignored

        preserveCase.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
```

```
                boolean preserveCase = ((Button) event.widget).getSelection();
                FileTreeLabelProvider ftlp = (FileTreeLabelProvider) tv
                        .getLabelProvider();
                ftlp.setPreserveCase(preserveCase);
            }
        });
        return composite;
    }

    public static void main(String[] args) {
        new FileBrowserTreeViewer().run();
    }
}

class FileTreeContentProvider implements ITreeContentProvider {

    public Object[] getChildren(Object arg0) {
        //返回树的下一级节点
        return ((File) arg0).listFiles();
    }

    public Object getParent(Object arg0) {

        //返回树的上一级节点
        return ((File) arg0).getParentFile();
    }

    public boolean hasChildren(Object arg0) {
        Object[] obj = getChildren(arg0);

        //判断树是否有下一级节点，true为在节点显示"+"信息
        return obj == null ? false : obj.length > 0;
    }

    public Object[] getElements(Object arg0) {

        //打印出树的输入信息，通常用户可以通过输入信息构建树
        System.out.println(arg0);
        // File.listRoots()作为树的根节点
        return File.listRoots();
    }

    public void dispose() {

    }

    public void inputChanged(Viewer arg0, Object arg1, Object arg2) {

    }
}
```

```
class FileTreeLabelProvider implements ILabelProvider {
    private List listeners;

    private Image file;

    private Image dir;

    boolean preserveCase;

    public FileTreeLabelProvider() {
        listeners = new ArrayList();

        try {
            //添加文件和目录的图标
            file = new Image(null, new FileInputStream("icons/file.gif"));
            dir = new Image(null, new FileInputStream("icons/directory.gif"));
        } catch (FileNotFoundException e) {

        }
    }

    public void setPreserveCase(boolean preserveCase) {
        this.preserveCase = preserveCase;

        //触发事件，重新得到显示标签
        LabelProviderChangedEvent event = new LabelProviderChangedEvent(this);
        for (int i = 0, n = listeners.size(); i < n; i++) {
            ILabelProviderListener ilpl = (ILabelProviderListener) listeners
                    .get(i);
            ilpl.labelProviderChanged(event);
        }
    }

    public Image getImage(Object arg0) {
        //返回目录或文件的图标
        return ((File) arg0).isDirectory() ? dir : file;
    }

    public String getText(Object arg0) {
        String text = ((File) arg0).getName();

        if (text.length() == 0) {
            text = ((File) arg0).getPath();
        }

        //返回目录或文件的文件名
        return preserveCase ? text : text.toUpperCase();
    }

    public void addListener(ILabelProviderListener arg0) {
        //添加监听器
```

```
            listeners.add(arg0);
        }

        public void dispose() {
            if (dir != null)
                dir.dispose();
            if (file != null)
                file.dispose();
        }

        public boolean isLabelProperty(Object arg0, String arg1) {
            return false;
        }

        public void removeListener(ILabelProviderListener arg0) {
            //删除监听器
            listeners.remove(arg0);
        }
    }
```

以上代码中，在主窗口上添加了 TreeViewer 组件，在树中显示当前磁盘和文件结构，另外可以通过“Preserve case”复选按钮选择是为原有显示文本还是全部大写显示文本，程序运行效果如图 9-1 所示。



图 9-1 TreeViewer 组件

9.2 TableViewer

JFace 中提供了 TableViewer 组件，用 TableViewer 来表示表格。TableViewer 和 TreeViewer 类似，它也提供了内容提供器（IStructuredContentProvider）和标签提供器（ITableLabelProvider），用来组织表格的信息。

9.2.1 TableViewer 构建步骤

TableViewer 中主要通过内容提供器（IStructuredContentProvider）和标签提供器（ITable-

LabelProvider）组织表中单元格的显示信息。TableViewer 构建步骤如下。

（1）创建 TableViewer 对象，例如“tableViewer = new TableViewer(table);”，其中 table 是 SWT 中 Table 对象。

（2）设定内容管理器，例如“tableViewer.setContentProvider(contentProvider);”。

（3）设定标签提供器，例如“tableViewer.setLabelProvider(labelProvider);”。

（4）设定 TableViewer 的输入数据，例如“tableViewer.setInput("root");”（用户可以通过输入数据构建表）。

另外，用户可以添加排序器和过滤器对表格数据进行排序和过滤，后面将通过实例介绍排序器和过滤器。

9.2.2 内容提供器（IStructuredContentProvider）

TableViewer 中的内容管理器比较简单，用户可以通过实现 getElements 返回表格的所有数据。

getElements 定义为“public Object[] getElements(Object inputElement)”，返加 Object[]对象数组，其中数组中每一个对象代表表格的一列。当 TableViewer 输入数据后，内容管理器将根据输入数据构造表格对象数据，如下代码所示。

```
public Object[] getElements(Object inputElement) {

    //通过输入的数据建表
    Vector v = (Vector) inputElement;
    return v.toArray();
}
```

其中，输入数据 inputElement 为 Vector 类型的变量，通过“v.toArray();”返回对象数组。

9.2.3 标签提供器（ITableLabelProvider）

TableViewer 中的标签提供器主要负责每个单元格中文本和图标的显示。ITableLabelProvider 接口要求实现 getColumnText 和 getColumnImage 两个主要的方法，getColumnText 返回指定单元格的显示文本，getColumnImage 返回指定单元格的显示图标，解释如下。

（1）getColumnText：此函数定义为“public String getColumnText(Object element, int columnIndex)”，其中 element 表示单元格所在行的对象，columnIndex 表示单元格所在的列，返回显示的文本。如下代码返回 Bug 对象的某特定列的显示文本。

```
public String getColumnText(Object element, int columnIndex) {
    Bug bug = (Bug) element;
    switch (columnIndex) {
    case 0:
        return bug.id;
    case 1:
        return bug.summary;
    case 2:
        return bug.assignedTo;
```

```
            case 3:
                return bug.isSolved ? "YES" : "NO";
            }
            return null;
        }
```

（2）getColumnImage：此函数定义为“public Image getColumnImage(Object element, int columnIndex)”，其中 element 表示单元格所在行的对象，columnIndex 表示单元格所在的列，返回此单元格的显示图标。如下代码返回 Bug 对象的某特定列的显示图标。

```
    public Image getColumnImage(Object element, int columnIndex) {
        if (columnIndex == 0)
            return bugIcon;
        return null;
    }
```

从上面代码可以看出，只有第一列显示图标，其他列不显示图标。

9.2.4 TableViewer 实例

本节的实例为一个 Bug 列表实例（BugTrackerTableViewer），程序的功能如下。

（1）在窗口中通过 ToolBar 建立工具栏，并通过 ToolBarManager 添加相应的 Action 到工具栏中。

（2）在表格中设置内容管理器和标签管理器。

（3）在表格的每一列添加排序器。当用户单击列头时，排序器将按列排序。排序器中主要通过 compare 方法比较两行（即两个对象）中的特定列单元格数据大小，返回比较后表格排列的顺序。

（4）通过工具栏的 Action(actionShowUnsolvedOnly)添加或删除过滤器，过滤器将根据过滤条件选择出符合要求的行（即对象）。

另外，程序中将在运行时加载上次存储的 Bug 列表，用户也可以保存当前的 Bug 列表，程序如例程 9-4 所示。

例程 9-4　BugTrackerTableViewer.java

```
package com.free.tableandtree;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Vector;

import org.eclipse.jface.action.Action;
import org.eclipse.jface.action.Separator;
import org.eclipse.jface.action.ToolBarManager;
import org.eclipse.jface.viewers.CellEditor;
```

```
import org.eclipse.jface.viewers.CheckboxCellEditor;
import org.eclipse.jface.viewers.ICellModifier;
import org.eclipse.jface.viewers.ILabelProviderListener;
import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.TextCellEditor;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.jface.viewers.ViewerFilter;
import org.eclipse.jface.viewers.ViewerSorter;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import org.eclipse.swt.widgets.Item;
import org.eclipse.swt.widgets.Listener;
import org.eclipse.swt.widgets.MessageBox;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Table;
import org.eclipse.swt.widgets.TableColumn;
import org.eclipse.swt.widgets.TableItem;
import org.eclipse.swt.widgets.ToolBar;

    public static class Bug {

        public String id;
        public String summary;
        public String assignedTo;
        public boolean isSolved;

        public Bug(String id, String summary, String assignedTo,
                boolean isSolved) {
            this.id = id;
            this.summary = summary;
            this.assignedTo = assignedTo;
            this.isSolved = isSolved;
        }

        public static Vector loadBugs(File file) {
            Vector v = new Vector();

            DataInputStream in = null;

            try {
                if (!file.exists())
                    return v;
                in = new DataInputStream(new FileInputStream(file));
```

```
            while (true) {
                String id = in.readUTF();
                String summary = in.readUTF();
                String assignedTo = in.readUTF();
                boolean solved = in.readBoolean();
                v.add(new Bug(id, summary, assignedTo, solved));
            }
        } catch (IOException ioe) {

        } finally {
            try {
                if (in != null)
                    in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return v;
    }
}

Display display = new Display();
Shell shell = new Shell(display);
Table table;
TableViewer tableViewer;
Vector bugs;
Image bugIcon = new Image(shell.getDisplay(), "icons/java2s.gif");
String[] colNames = new String[] { "ID", "Summary", "Assigned to", "Solved" };

//实现表格的排序器
class BugSorter extends ViewerSorter {
    private String property;
    private int propertyIndex;

    public BugSorter(String sortByProperty) {
        for (int i = 0; i < colNames.length; i++) {
            if (colNames[i].equals(sortByProperty)) {
                this.property = sortByProperty;
                this.propertyIndex = i;
                return;
            }
        }
        throw new IllegalArgumentException("Unrecognized property: "
                + sortByProperty);
    }

    //比较e1和e2的大小
    public int compare(Viewer viewer, Object e1, Object e2) {
        Bug bug1 = (Bug) e1;
        Bug bug2 = (Bug) e2;
        switch (propertyIndex) {
```

```
                case 0:
                    return bug1.id.compareTo(bug2.id);
                case 1:
                    return bug1.summary.compareTo(bug2.summary);
                case 2:
                    return bug1.assignedTo.compareTo(bug2.assignedTo);
                case 3:
                    if (bug1.isSolved == bug2.isSolved)
                        return 0;
                    if (bug1.isSolved)
                        return 1;
                    else
                        return -1;
                default:
                    return 0;
                }
            }

        }

        public BugTrackerTableViewer() {
            // 添加工具栏的相应Action
            //添加新建按钮的Action
            Action actionAddNew = new Action("New bug") {
                public void run() {
                    // Append.
                    Bug bug = new Bug("", "", "", false);
                    bugs.add(bug);
                    tableViewer.refresh(false);
                }
            };

            Action actionDelete = new Action("Delete selected") {
                public void run() {
                    IStructuredSelection selection = (IStructuredSelection)
                    tableViewer.getSelection();
                    Bug bug = (Bug) selection.getFirstElement();
                    if (bug == null) {
                        System.out.println("Please select an item first. ");
                        return;
                    }
                    MessageBox messageBox = new MessageBox(shell, SWT.YES | SWT.NO);
                    messageBox.setText("Confirmation");
                    messageBox.setMessage("Are you sure to remove the bug with id #"
                                          + bug.id);
                    if (messageBox.open() == SWT.YES) {
                        bugs.remove(bug);
                        tableViewer.refresh(false);
                    }
                }
            };
```

```
Action actionSave = new Action("Save") {
    public void run() {
        saveBugs(bugs);
    }
};
//添加表的过滤器
final ViewerFilter filter = new ViewerFilter() {
    public boolean select(Viewer viewer, Object parentElement,
            Object element) {
        if (!((Bug) element).isSolved)
            return true;
        return false;
    }
};
Action actionShowUnsolvedOnly = new Action("Show unsolved only") {
    public void run() {
        if (!isChecked())
            tableViewer.removeFilter(filter);
        else
            tableViewer.addFilter(filter);
    }
};
actionShowUnsolvedOnly.setChecked(false);

ToolBar toolBar = new ToolBar(shell, SWT.RIGHT | SWT.FLAT);

ToolBarManager manager = new ToolBarManager(toolBar);
// 添加工具项，并把工具项关联到相应的Action
manager.add(actionAddNew);
manager.add(actionDelete);
manager.add(new Separator());
manager.add(actionSave);
manager.add(new Separator());
manager.add(actionShowUnsolvedOnly);
manager.update(true);
shell.setLayout(new GridLayout());
table = new Table(shell, SWT.BORDER | SWT.FULL_SELECTION);

TableColumn tcID = new TableColumn(table, SWT.LEFT);
tcID.setText(colNames[0]);

TableColumn tcSummary = new TableColumn(table, SWT.NULL);
tcSummary.setText(colNames[1]);

TableColumn tcAssignedTo = new TableColumn(table, SWT.NULL);
tcAssignedTo.setText(colNames[2]);

TableColumn tcSolved = new TableColumn(table, SWT.NULL);
tcSolved.setText(colNames[3]);
```

```
tcID.setWidth(60);
tcSummary.setWidth(200);
tcAssignedTo.setWidth(80);
tcSolved.setWidth(50);
tableViewer = new TableViewer(table);
tableViewer.getTable().setLinesVisible(true);
tableViewer.getTable().setHeaderVisible(true);
tableViewer.getTable().setLayoutData(new GridData(GridData.FILL_BOTH));
//设定表的内容管理器
tableViewer.setContentProvider(new IStructuredContentProvider() {
    //获得所有对象
    public Object[] getElements(Object inputElement) {

        //通过输入的数据建表
        Vector v = (Vector) inputElement;
        return v.toArray();
    }

    public void dispose() {
        System.out.println("Disposing ...");
    }

    public void inputChanged(Viewer viewer, Object oldInput,
            Object newInput) {
        System.out.println("Input changed: old=" + oldInput + ", new="
                + newInput);
    }
});
//设定表的标签管理器
tableViewer.setLabelProvider(new ITableLabelProvider() {
    public Image getColumnImage(Object element, int columnIndex) {
        if (columnIndex == 0)
            return bugIcon;
        return null;
    }

    //获得列的显示文本
    public String getColumnText(Object element, int columnIndex) {
        Bug bug = (Bug) element;
        switch (columnIndex) {
        case 0:
            return bug.id;
        case 1:
            return bug.summary;
        case 2:
            return bug.assignedTo;
        case 3:
            return bug.isSolved ? "YES" : "NO";
        }
        return null;
    }
```

```
        public void addListener(ILabelProviderListener listener) {
        }

        public void dispose() {
        }

        public boolean isLabelProperty(Object element, String property) {
            return false;
        }

        public void removeListener(ILabelProviderListener listener) {
        }
    });
    //设定单元格编辑器
    tableViewer.setColumnProperties(colNames);
    CellEditor[] cellEditors = new CellEditor[4];
    cellEditors[0] = new TextCellEditor(table);
    cellEditors[1] = cellEditors[0];
    cellEditors[2] = cellEditors[0];
    cellEditors[3] = new CheckboxCellEditor(table);
    tableViewer.setCellEditors(cellEditors);
    //设定单元格修改器
    tableViewer.setCellModifier(new ICellModifier() {
        public boolean canModify(Object element, String property) {
            return true;
        }

        public Object getValue(Object element, String property) {

            int index = -1;
            for (int i = 0; i < colNames.length; i++) {
                if (colNames[i].equals(property)) {
                    index = i;
                    break;
                }
            }
            Bug bug = (Bug) element;
            switch (index) {
            case 0:
                return bug.id;
            case 1:
                return bug.summary;
            case 2:
                return bug.assignedTo;
            case 3:
                return new Boolean(bug.isSolved);
            }
            return null;
        }
```

```
        public void modify(Object element, String property, Object value) {
            System.out.println("Modify: " + element + ", " + property
                    + ", " + value);

            int index = -1;
            for (int i = 0; i < colNames.length; i++) {
                if (colNames[i].equals(property)) {
                    index = i;
                    break;
                }
            }
            Bug bug = null;
            if (element instanceof Item) {
                TableItem item = (TableItem) element;
                bug = (Bug) item.getData();
            } else {
                bug = (Bug) element;
            }
            switch (index) {
            case 0:
                bug.id = (String) value;
                break;
            case 1:
                bug.summary = (String) value;
                break;
            case 2:
                bug.assignedTo = (String) value;
                break;
            case 3:
                bug.isSolved = ((Boolean) value).booleanValue();
                break;
            }
            tableViewer.update(bug, null);
        }
    });
    //设定表格的排序方式
    //单击第一列列头按第一列排序
    tcID.addListener(SWT.Selection, new Listener() {
        public void handleEvent(Event event) {
            tableViewer.setSorter(new BugSorter(colNames[0]));
        }
    });
    //单击第二列列头按第二列排序
    tcSummary.addListener(SWT.Selection, new Listener() {
        public void handleEvent(Event event) {
            tableViewer.setSorter(new BugSorter(colNames[1]));
        }
    });
    //单击第三列列头按第三列排序
    tcAssignedTo.addListener(SWT.Selection, new Listener() {
        public void handleEvent(Event event) {
```

```
            tableViewer.setSorter(new BugSorter(colNames[2]));
        }
    });
    //单击第四列列头按第四列排序
    tcSolved.addListener(SWT.Selection, new Listener() {
        public void handleEvent(Event event) {
            tableViewer.setSorter(new BugSorter(colNames[3]));
        }
    });
    //从文件中读取Bug记录
    bugs = Bug.loadBugs(new File("bugs.dat"));
    //把bugs作为表格的输入
    tableViewer.setInput(bugs);

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) {
            display.sleep();
        }
    }

    display.dispose();
}

//保存当前的Bug记录
private void saveBugs(Vector v) {

    DataOutputStream out = null;
    try {
        File file = new File("bugs.dat");

        out = new DataOutputStream(new FileOutputStream(file));

        for (int i = 0; i < v.size(); i++) {
            Bug bug = (Bug) v.elementAt(i);
            out.writeUTF(bug.id);
            out.writeUTF(bug.summary);
            out.writeUTF(bug.assignedTo);
            out.writeBoolean(bug.isSolved);
        }
    } catch (IOException ioe) {

    } finally {
        try {
            if (out != null)
                out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
            }
        }

        public static void main(String[] args) {
            new BugTrackerTableViewer();
        }
    }
```

上例为表格添加了过滤器和排序器，用户可以单击表格的列头按此列进行排序。另外，还能够通过工具栏按钮过滤掉已解决的 Bug 信息。程序运行效果如图 9-2 所示。

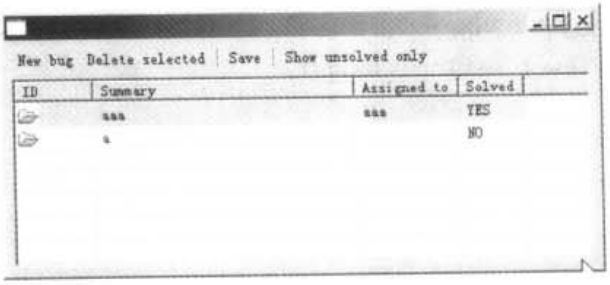


图 9-2　TableViewer 组件

9.3　ListViewer

JFace 中提供了 ListViewer 组件，用来表示列表框。关于 ListViewer 的介绍如下。

9.3.1　ListViewer 功能

JFace 中提供了 ListViewer 以对列表进行管理。ListViewer 也有内容管理器（IStructured ContentProvider）和标签管理器（ILabelProvider），功能和 TreeViewer、TableViewer 的功能基本上是一样的。

另外，ListViewer 、TreeViewer 和 TableViewer 都是直接或间接从 StructuredViewer 类继承过来的，StructuredViewer 实现了一些公用的方法。

（1）setSorter：定义为“public void setSorter(ViewerSorter sorter)”，用户可以通过 setSorter 设置 Viewer 的排序器，当然也可以通过 getSorter 得到排序器。

（2）addFilter：定义为“public void addFilter(ViewerFilter filter)”，用户可以通过 addFilter 添加过滤器，在 Viewer 中过滤器可以同时有多个，可以通过 getFilters 得到过滤器列表。

（3）getSelection：定义为“public ISelection getSelection()”，返回选择的节点对象，用户可以通过 ISelection 得到当前选择的对象（可参考 TableViewer 的实例）。

9.3.2　ListViewer 实例

本节的实例为一个食品的 ListViewer 列表，另外还添加了一个过滤器，用户通过“Show only healthy”复选框选择健康食品，代码如例程 9-5 所示。

例程 9-5 FoodListViewer.java

```
package com.free.tableandtree;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.eclipse.jface.viewers.ILabelProvider;
import org.eclipse.jface.viewers.ILabelProviderListener;
import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.ListViewer;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.jface.viewers.ViewerFilter;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class FoodListViewer extends ApplicationWindow {

    private HealthyFilter filter = new HealthyFilter();

    public FoodListViewer() {
        super(null);
    }

    public void run() {
        setBlockOnOpen(true);

        open();

        Display.getCurrent().dispose();
    }

protected void configureShell(Shell shell) {
        super.configureShell(shell);
        shell.setText("Food List");
    }

    protected Control createContents(Composite parent) {
        Composite composite = new Composite(parent, SWT.NONE);
        composite.setLayout(new GridLayout(1, false));
```

```
        Button filterHealthy = new Button(composite, SWT.CHECK);
        filterHealthy.setText("&Show only healthy");

        // 添加ListViewer组件
        final ListViewer lv = new ListViewer(composite);
        // 设置内容管理器
        lv.setContentProvider(new FoodContentProvider());
        // 设置标签管理器
        lv.setLabelProvider(new FoodLabelProvider());
        // 设置ListViewer的输入信息
        lv.setInput(new GroceryList());

        filterHealthy.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                if (((Button) event.widget).getSelection())
                    // 如果filterHealthy为选中状态，添加过滤器
                    lv.addFilter(filter);
                else
                    // 如果filterHealthy不为选中状态，删除过滤器
                    lv.removeFilter(filter);
            }
        });

        parent.pack();
        return composite;
    }

    public static void main(String[] args) {
        new FoodListViewer().run();
    }
}

class HealthyFilter extends ViewerFilter {

    public boolean select(Viewer arg0, Object arg1, Object arg2) {
        //选择健康食品
        return ((Food) arg2).isHealthy();
    }
}

class Food {

    private String name;

    private boolean healthy;

    public Food(String name, boolean healthy) {
        this.name = name;
        this.healthy = healthy;
    }
```

```
    public boolean isHealthy() {
        return healthy;
    }

    public String getName() {
        return name;
    }
}

class FoodLabelProvider implements ILabelProvider {

    public Image getImage(Object arg0) {
        return null;
    }

    public String getText(Object arg0) {
        return ((Food) arg0).getName();
    }

    public void addListener(ILabelProviderListener arg0) {

    }

    public void dispose() {

    }

    public boolean isLabelProperty(Object arg0, String arg1) {
        return false;
    }

    public void removeListener(ILabelProviderListener arg0) {

    }
}

class GroceryList {
    private List foods;

    public GroceryList() {
        foods = new ArrayList();

        //添加食品列表
        foods.add(new Food("Broccoli", true));
        foods.add(new Food("Bundt Cake", false));
        foods.add(new Food("Cabbage", true));
        foods.add(new Food("Candy Canes", false));
        foods.add(new Food("Eggs", true));
        foods.add(new Food("Potato Chips", false));
        foods.add(new Food("Milk", true));
        foods.add(new Food("Soda", false));
```

```
            foods.add(new Food("Chicken", true));
            foods.add(new Food("Cinnamon Rolls", false));
        }

        public List getFoods() {
            return Collections.unmodifiableList(foods);
        }
    }

    class FoodContentProvider implements IStructuredContentProvider {
        public Object[] getElements(Object arg0) {
            //返回所有食品列表的数组
            return ((GroceryList) arg0).getFoods().toArray();
        }

        public void dispose() {

        }

        public void inputChanged(Viewer arg0, Object arg1, Object arg2) {

        }
    }
```

本例添加了一个食品列表，程序运行效果如图 9-3 所示。



图 9-3　ListViewer 组件

9.4　小　　结

本节介绍了 JFace 的 TreeViewer、TableViewer 和 ListViewer 组件，这几个都是比较常用的组件，TreeViewer、TableViewer 和 ListViewer 是通过封装 SWT 中的 Tree、Table 和 List 实现的。

另外，JFace 中的 Viewer 组件都是通过相应的内容管理器和标签管理器来实现的。通过学习本章读者应该掌握这两个管理器的实现。

第10章 对 话 框

对话框是 GUI 开发中不可缺少的一部分，对话框有系统定义的对话框、消息对话框和自定义对话框。

本章将介绍如下几种类型的对话框。

- 消息对话框
- 输入对话框
- 进度监视对话框
- 自定义对话框
- 常用系统对话框

10.1 消息对话框

MessageDialog 实现了消息对话框，具体介绍如下。

10.1.1 消息对话框的功能

消息对话框（MessageDialog）是比较常用的对话框，它能够提供便捷的人机交互功能。JFace 中提供了 MessageDialog 来实现消息对话框，它能够提供几种风格的显示。

（1）确认对话框：MessageDialog 通过 openConfirm 方法打开确认对话框，例如“MessageDialog.openConfirm(shell, "Confirm", text.getText());”。

（2）错误对话框：MessageDialog 通过 openError 方法打开错误对话框，例如“MessageDialog.openError(shell, "Error", text.getText());”。

（3）提示信息对话框：MessageDialog 通过 openInformation 方法打开提示信息对话框，例如“MessageDialog.openInformation(shell, "Information", text.getText());”。

（4）提问对话框：MessageDialog 通过 openQuestion 方法打开提问对话框，例如“MessageDialog.openQuestion(shell, "Question", text.getText());”。

（5）警告对话框：MessageDialog 通过 openWarning 方法打开警告对话框，例如“MessageDialog.openWarning(shell, "Warning", text.getText());”。

其中，这几个方法都是静态的方法，且参数都是一样的，第一个参数表示父窗口组件，第二个参数表示标题信息，第三个参数表示显示的信息。

另外，openConfirm 返回 boolean 型的值，表示用户是否选择确认，openQuestion 也返回 boolean 型的值，表示是否正确。

10.1.2 消息对话框实例

创建消息对话框比较简单，本节将通过实例演示每种风格的消息对话框，代码如例程 10-1 所示。

例程 10-1 MessageDialogExample.java

```
package com.free.dialog;

import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class MessageDialogExample extends ApplicationWindow {
    public MessageDialogExample() {
        super(null);
    }

    public void run() {
        setBlockOnOpen(true);

        open();

        Display.getCurrent().dispose();
    }

    protected void configureShell(Shell shell) {
        super.configureShell(shell);

        shell.setText("Send Message");
        shell.setSize(500, 400);
    }

    protected Control createContents(Composite parent) {
        Composite composite = new Composite(parent, SWT.NONE);
        composite.setLayout(new GridLayout(5, true));

        final Text text = new Text(composite, SWT.MULTI | SWT.BORDER
                | SWT.V_SCROLL);
```

```
GridData data = new GridData(GridData.FILL_BOTH);
data.horizontalSpan = 5;
text.setLayoutData(data);

Button confirm = new Button(composite, SWT.PUSH);
confirm.setText("Confirm");
confirm.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

Button error = new Button(composite, SWT.PUSH);
error.setText("Error");
error.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

Button information = new Button(composite, SWT.PUSH);
information.setText("Information");
information.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

Button question = new Button(composite, SWT.PUSH);
question.setText("Question");
question.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

Button warning = new Button(composite, SWT.PUSH);
warning.setText("Warning");
warning.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

final Label label = new Label(composite, SWT.NONE);
data = new GridData(GridData.FILL_HORIZONTAL);
data.horizontalSpan = 5;
label.setLayoutData(data);

final Shell shell = parent.getShell();

confirm.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        //显示确认对话框
        boolean b = MessageDialog.openConfirm(shell, "Confirm", text
                .getText());
        //返回是个单击确定按钮 (OK)
        label.setText("Returned " + Boolean.toString(b));
    }
});

error.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
        //显示错误对话框
        MessageDialog.openError(shell, "Error", text.getText());
        label.setText("Returned void");
    }
});

information.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent event) {
```

```
                //显示提示信息对话框
                MessageDialog.openInformation(shell, "Information", text
                        .getText());
                label.setText("Returned void");
            }
        });

        question.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                //显示提问对话框
                boolean b = MessageDialog.openQuestion(shell, "Question", text
                        .getText());
                //返回是否对的按钮（Yes）
                label.setText("Returned " + Boolean.toString(b));
            }
        });

        warning.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                //显示警告对话框
                MessageDialog.openWarning(shell, "Warning", text.getText());
                label.setText("Returned void");
            }
        });

        return composite;
    }

    public static void main(String[] args) {
        new MessageDialogExample().run();
    }
}
```

以上代码演示了如何创建每种形式的对话框，程序运行效果如图 10-1 所示。

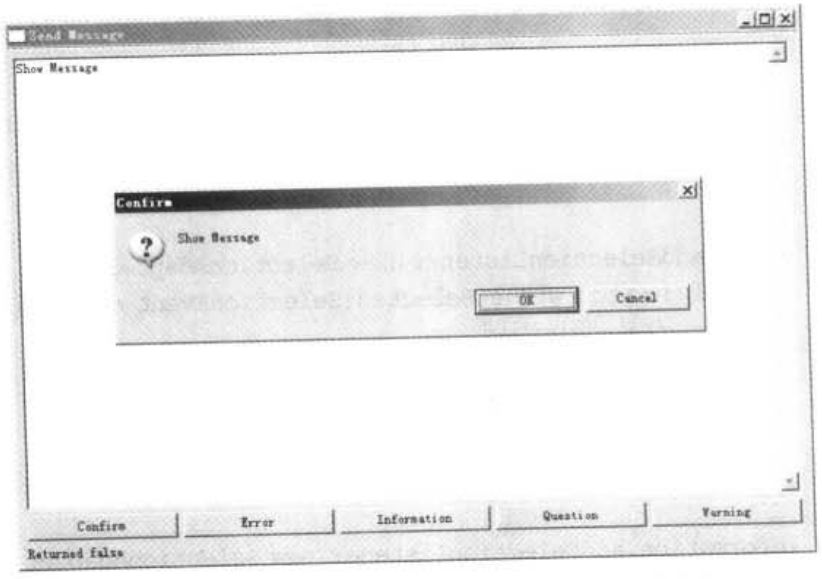


图 10-1　消息对话框

10.2 输入对话框

InputDialog 实现了输入对话框，具体介绍如下。

10.2.1 输入对话框的功能

新建输入对话框（InputDialog）要创建一个 InputDialog 实例。InputDialog 构造函数为："public InputDialog(Shell parentShell, String dialogTitle,String dialogMessage, String initialValue, IInputValidator validator)"，解释如下。

- parentShell：表示输入框的父窗口。
- dialogTitle：表示输入框的标题。
- dialogMessage：表示输入框的提示信息。
- initialValue：表示初始输入框的值。
- validator：表示输入框的校验器。

校验器是一个 IInputValidator 类型的对象（也可以为空），校验器中要实现 isValid 方法，该方法返回一个 String 类型的值或 null，返回值为 String 类型时为错误的提示信息，如果为 null 则表示校验正确。下面代码为 isValid 的实现。

```
public String isValid(String newText) {
  int len = newText.length();

  if (len < 5) return "Too short";
  if (len > 8) return "Too long";

  return null;
}
```

上面代码表示如果newText的长度大于8或小于5时校验出错，提示错误信息为"Too short"或"Too long"，窗口的确定按钮为禁用。

10.2.2 输入对话框实例

为了更好地理解输入对话框，下面通过实例来介绍。实例中新建一个输入对话框，并加入了校验器，代码如例程 10-2 所示。

例程 10-2 InputDialogExample.java

```
package com.free.dialog;

import org.eclipse.jface.dialogs.InputDialog;
import org.eclipse.jface.dialogs.IInputValidator;
import org.eclipse.jface.window.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.*;
```

```
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class InputDialogExample extends ApplicationWindow {

    public InputDialogExample() {
        super(null);
    }

    public void run() {
        setBlockOnOpen(true);

        open();

        Display.getCurrent().dispose();
    }

    protected void configureShell(Shell shell) {
        super.configureShell(shell);

        shell.setText("Get Input");
    }

    protected Control createContents(Composite parent) {
        Composite composite = new Composite(parent, SWT.NONE);
        composite.setLayout(new GridLayout(1, false));

        final Label label = new Label(composite, SWT.NONE);
        label.setText("This will display the user input from InputDialog");

        Button show = new Button(composite, SWT.PUSH);
        show.setText("Get Input");
        show.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {

                //新建输入对话框，并添加校验器
                InputDialog dlg = new InputDialog(Display.getCurrent()
                        .getActiveShell(), "", "Enter 5-8 characters", label
                        .getText(), new LengthValidator());
                //打开输入对话框
                if (dlg.open() == Window.OK) {
                    // 如果用户单击OK按钮，则给标签赋值
                    label.setText(dlg.getValue());
                }
            }
        });

        parent.pack();
        return composite;
    }
```

```
    public static void main(String[] args) {
        new InputDialogExample().run();
    }
}

class LengthValidator implements IInputValidator {
    //校验用户的输入，newText 为对话框的输入值
    public String isValid(String newText) {
        int len = newText.length();

        if (len < 5)
            return "Too short";
        if (len > 8)
            return "Too long";

        //输入数据正确
        return null;
    }
}
```

上例为输入对话框添加了一个校验器，程序运行效果如图 10-2 所示。图中输入数据“very good”太长，校验未通过，确定按钮（OK）为禁用。

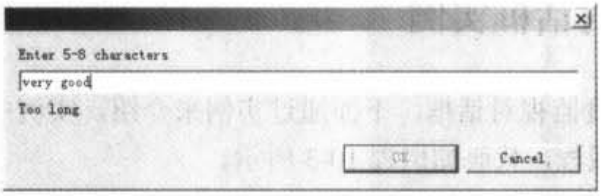


图 10-2 输入对话框

10.3 进度监视对话框

ProgressMonitorDialog 实现了进度监视对话框，具体介绍如下。

10.3.1 进度监视对话框的功能

创建进度监视对话框（ProgressMonitorDialog）要新建一个 ProgressMonitorDialog 实例。ProgressMonitorDialog 的构造函数参数为父窗口 Shell，例如“new ProgressMonitorDialog(shell)”。

要运行 ProgressMonitorDialog 需要调用 ProgressMonitorDialog 的 run 方法。run 方法定义为“public void run(boolean fork, boolean cancelable, IRunnableWithProgress runnable) throws InvocationTargetException, InterruptedException”，解释如下。

- fork：表示当进度对话框运行时，是否刷新界面。
- cancelable：表示对话框是否可以取消。
- runnable：表示进度处理线程。

另外，run 方法可能会抛出“InvocationTargetException, InterruptedException”异常，用户

可以通过 IRunnableWithProgress 接口实现 run 方法对进度监视对话框中进度条的控制，如下代码所示。

```
public void run(IProgressMonitor monitor) throws InvocationTargetException,
   InterruptedException {
    monitor.beginTask("Running long running operation",100);
          for (int total = 0; total < 50; total++) {
           Thread.sleep(100);
           monitor.worked(2);
          }

  monitor.done();

 }
```

代码解释如下。

- beginTask：monitor.beginTask 方法表示进度监视对话框开始执行，第一个参数为进度监视对话框的提示信息，第二个参数为把进度分成多少份。
- worked：monitor.worked 方法表示进度每次完成多少份（进度条会相应增加）。
- done：monitor.done 方法表示进度监视对话框结束。

10.3.2 进度监视对话框实例

为了更好地理解进度监视对话框，下面通过实例来介绍，实例中新建一个进度监视对话框，并加入了进度处理线程，代码如例程 10-3 所示。

例程 10-3 ProgressMonitorDialogExample.java

```
package com.free.dialog;

import java.lang.reflect.InvocationTargetException;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.dialogs.ProgressMonitorDialog;
import org.eclipse.jface.operation.IRunnableWithProgress;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class ProgressMonitorDialogExample extends ApplicationWindow {
```

```
public ProgressMonitorDialogExample() {
    super(null);
}

public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
}

protected void configureShell(Shell shell) {
    super.configureShell(shell);

    shell.setText("Show Progress");
}

protected Control createContents(Composite parent) {
    Composite composite = new Composite(parent, SWT.NONE);
    composite.setLayout(new GridLayout(1, true));

    final Button indeterminate = new Button(composite, SWT.CHECK);
    indeterminate.setText("Indeterminate");

    Button showProgress = new Button(composite, SWT.NONE);
    showProgress.setText("Show Progress");

    final Shell shell = parent.getShell();

    //添加事件
    showProgress.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent event) {
            try {
                //新建进度监视对话框，并运行
                new ProgressMonitorDialog(shell).run(true, true,
                        new LongRunningOperation(indeterminate
                                .getSelection()));
            } catch (InvocationTargetException e) {
                MessageDialog.openError(shell, "Error", e.getMessage());
            } catch (InterruptedException e) {
                MessageDialog.openInformation(shell, "Cancelled",
                        e.getMessage());
            }
        }
    });

    parent.pack();
    return composite;
}

public static void main(String[] args) {
    new ProgressMonitorDialogExample().run();
```

```
        }
    }

    class LongRunningOperation implements IRunnableWithProgress {
        private static final int TOTAL_TIME = 10000;

        private static final int INCREMENT = 500;

        private boolean indeterminate;

        public LongRunningOperation(boolean indeterminate) {
            this.indeterminate = indeterminate;
        }

        public void run(IProgressMonitor monitor) throws InvocationTargetException,
                InterruptedException {

            //开始处理进度，如果 indeterminate 为 false，整个进度为 TOTAL_TIME = 10000
            //否则，进度条为自动增加的
            monitor.beginTask("Running long running operation",
                    indeterminate ? IProgressMonitor.UNKNOWN : TOTAL_TIME);
            for (int total = 0; total < TOTAL_TIME && !monitor.isCanceled(); total
                    += INCREMENT) {
                Thread.sleep(INCREMENT);
                monitor.worked(INCREMENT);
                //到达一半的时间显示相应的信息
                if (total == TOTAL_TIME / 2)
                    monitor.subTask("Doing second half");
            }
            //完成任务
            monitor.done();
            if (monitor.isCanceled())
                throw new InterruptedException(
                        "The long running operation was cancelled");
        }
    }
```

程序中通过线程控制进度对话框的运行进度，程序运行效果如图 10-3 所示。

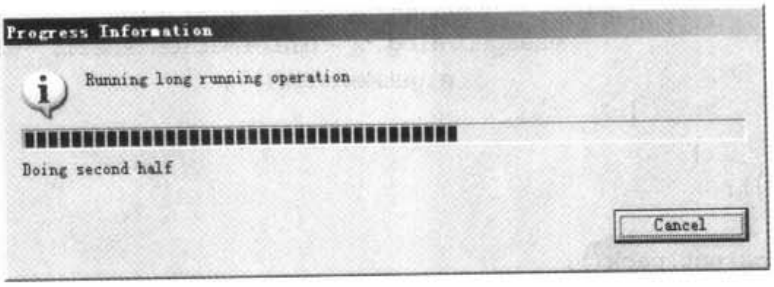


图 10-3　进度监视对话框

10.4 自定义对话框

在界面的开发中，并不是所有的对话框都会符合用户的要求，用户经常要自己定义对话框。要定义自己的对话框，可以通过继承 Dialog 实现，也可以通过 Dialog 的子类实现，本节将继承 JFace 的 TitleAreaDialog 实现自定义对话框。

10.4.1 带标题区域对话框

带标题区域对话框（TitleAreaDialog）继承于 JFace 的 Dialog 类，TitleAreaDialog 实现了几个比较常用的方法。

- createContents：重写了 Dialog 的 createContents 方法，把对话框区域分成标题区域和对话框区域。
- createTitleArea：建立标题区域的组件。
- createDialogArea：建立对话框区域的组件。
- setTitle：设置显示的标题。
- setMessage：设置显示的信息。
- setErrorMessage：设置显示的错误信息。

10.4.2 自定义对话框实例

本节实例中将重写 TitleAreaDialog 的 createDialogArea 方法和重写 Dialog 的 createButtonsForButtonBar 方法，创建对话框区域和创建对话框按钮，并且添加对话框的校验方法 validate，代码如例程 10-4 所示。

例程 10-4 CreateOwnDialogExample.java

```
package com.free.dialog;

import org.eclipse.jface.dialogs.TitleAreaDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.List;
import org.eclipse.swt.widgets.Shell;

public class CreateOwnDialogExample {
```

```
    public static void main(String[] args) {
        Display display = new Display();
           final Shell shell = new Shell(display);

           shell.setSize(400,100);
           shell.setLayout(new FillLayout());

           Button b1 = new Button(shell, SWT.PUSH);
           b1.setText("Open mail Dialog ...");
           b1.pack();
           b1.setLocation(50, 50);
           b1.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent se) {

              MailDialog dial = new MailDialog(shell, new String[]{"mail", "gan"});
              dial.open();
            }
           });
           shell.open();
           while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
              display.sleep();
           }
           display.dispose();
         }

}

class MailDialog extends TitleAreaDialog {
      public static final int OPEN = 9999;

      public static final int DELETE = 9998;

      List list;

      String[] items;

      String[] itemsToOpen;

      public MailDialog(Shell shell, String[] items) {
        super(shell);
        this.items = items;
      }

      public void create() {
        super.create();
        setTitle("Mail");
        setMessage("You have mail! \n It could be vital for this evening...");
      }

      //重写createDialogArea 方法，创建对话框区域
```

```
protected Control createDialogArea(Composite parent) {
  //添加对话框区域的父面板
  final Composite area = new Composite(parent, SWT.NULL);
  final GridLayout gridLayout = new GridLayout();
  gridLayout.marginWidth = 15;
  gridLayout.marginHeight = 10;
  //设置父面板的布局方式
  area.setLayout(gridLayout);
  //在对话框区域中添加List组件
  list = new List(area, SWT.BORDER | SWT.MULTI);
  final GridData gridData = new GridData();
  gridData.widthHint = 200;
  list.setLayoutData(gridData);
  list.addSelectionListener(new SelectionAdapter() {
   public void widgetSelected(SelectionEvent e) {
    validate();
   }
  });
  for (int i = 0; i < items.length; i++) {
   list.add(items[i]);
  }
  return area;
}

//添加自定义对话框的校验方法
private void validate() {
  boolean selected = (list.getSelectionCount() > 0);
  getButton(OPEN).setEnabled(selected);
  getButton(DELETE).setEnabled(selected);
  if (!selected)
   setErrorMessage("Select at least one entry!");
  else
   setErrorMessage(null);
}

//重写Dialog的createButtonsForButtonBar方法创建对话框按钮
protected void createButtonsForButtonBar(Composite parent) {
  //添加Open按钮
  Button openButton = createButton(parent, OPEN, "Open", true);
  // 初始状态为禁用
  openButton.setEnabled(false);
  //为Open按钮添加事件
  openButton.addSelectionListener(new SelectionAdapter() {
   public void widgetSelected(SelectionEvent e) {
    itemsToOpen = list.getSelection();
    setReturnCode(OPEN);
    close();
   }
  });
  //添加delete按钮
  Button deleteButton = createButton(parent, DELETE, "Delete", false);
```

```
        deleteButton.setEnabled(false);
        //为delete按钮添加事件
        deleteButton.addSelectionListener(new SelectionAdapter() {
          public void widgetSelected(SelectionEvent e) {
            int selectedItems[] = list.getSelectionIndices();
            list.remove(selectedItems);
            //校验当前数据
            validate();
          }
        });
        Button cancelButton = createButton(parent, CANCEL, "Cancel", false);
        //添加cancel按钮
        cancelButton.addSelectionListener(new SelectionAdapter() {
          public void widgetSelected(SelectionEvent e) {
            setReturnCode(CANCEL);
            close();
          }
        });
      }

      public String[] getItemsToOpen() {
        return itemsToOpen;
      }

    }
```

上例添加了一个自定义对话框，并在测试程序中新建了自定义对话框，程序运行效果如图 10-4 所示。

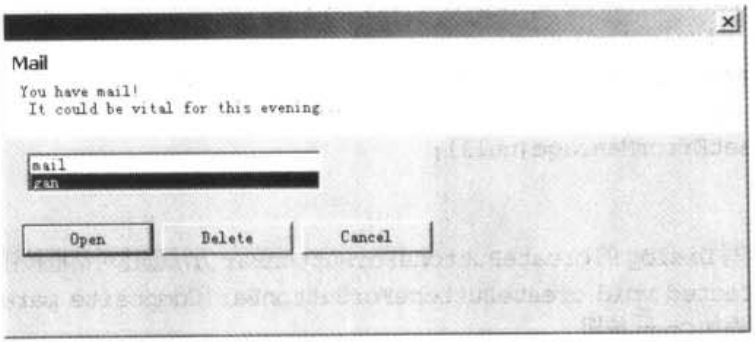


图 10-4　自定义对话框

一般来说，用户可以通过继承 JFace 现有的对话框进行扩展，在现有对话框的基础上可以非常容易地定义自己的对话框。

10.5　常用系统对话框

在 SWT 的开发过程中，用户还会经常使用到打印对话框、颜色对话框、字体对话框、文件对话框和目录对话框等对话框，本节将介绍这些对话框的使用。另外，每种类型的对话框都可以在本书源代码中找到实例。

10.5.1 打印对话框（PrintDialog）

打印对话框中用户能选择打印机和设置打印的一些选项，如图 10-5 所示。

新建打印对话框通过 PrintDialog，例如“PrintDialog printDialog = new PrintDialog(s, SWT.NONE);”，另外，要打印相关的数据可以通过 Printer 对象（如有兴趣可参考本书源代码）。

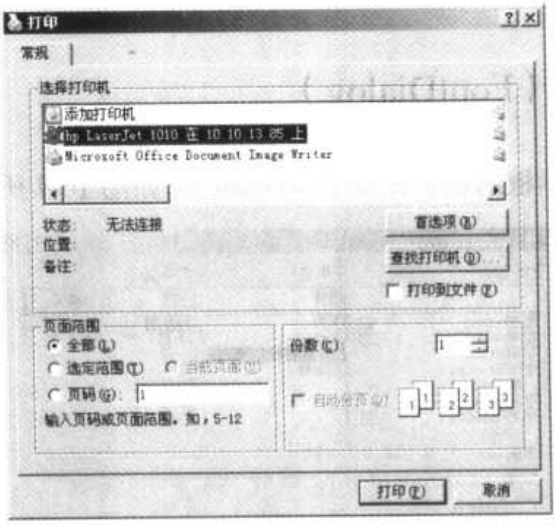


图 10-5 打印对话框

10.5.2 颜色对话框（ColorDialog）

当用户要改变颜色的设置，就要用到颜色对话框，如图 10-6 所示。

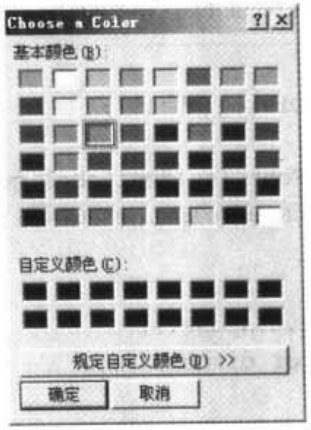


图 10-6 颜色对话框

用户可以通过如下代码片段新建颜色对话框，并设置标签的颜色。

```
//新建颜色对话框
ColorDialog dlg = new ColorDialog(shell);
//设置默认的颜色
dlg.setRGB(colorLabel.getBackground().getRGB());
//设置标题字体
dlg.setText("Choose a Color");
```

```
//打开对话框
RGB rgb = dlg.open();
if (rgb != null) {
  color.dispose();
  color = new Color(shell.getDisplay(), rgb);
  //设置标签的颜色
  colorLabel.setBackground(color);
}
```

10.5.3 字体对话框（FontDialog）

用户可以通过字体对话框选择字体及字体的颜色，如图 10-7 所示。

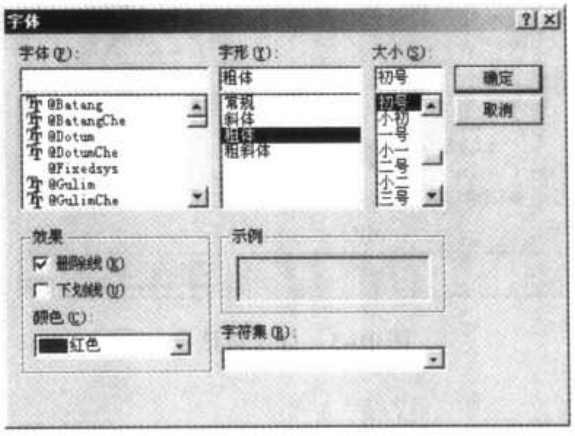


图 10-7 字体对话框

新建字体对话框比较简单，如下代码片段演示如何设置字体对话框的初始字体和颜色，以及选择了相应的字体和颜色后设置标签的字体和颜色。

```
// 新建字体对话框
FontDialog dlg = new FontDialog(shell);
//设置初始字体和颜色
if (font != null) dlg.setFontList(fontLabel.getFont().getFontData());
if (color != null) dlg.setRGB(color.getRGB());
//打开对话框
if (dlg.open() != null) {
    if (font != null) font.dispose();
    if (color != null) color.dispose();

    //新建字体对象
    font = new Font(shell.getDisplay(), dlg.getFontList());
    //设置标签的字体
    fontLabel.setFont(font);

    //新建颜色对象
    color = new Color(shell.getDisplay(), dlg.getRGB());
    //设置标签颜色
    fontLabel.setForeground(color);
```

10.5.4 文件对话框（FileDialog）

文件对话框是比较常用的对话框，当用户要从磁盘选择文件时将用文件对话框选取文件，如图 10-8 所示。

新建文件对话框，用户可以设置对话框的初始路径、标题和扩展名过滤等属性，如下代码片段所示。

```
//新建文件对话框，并设置为打开的方式
FileDialog fd = new FileDialog(s, SWT.OPEN);
//设置文件对话框的标题
fd.setText("Open");
//设置初始路径
fd.setFilterPath("C:/");
//设置扩展名过滤
String[] filterExt = { "*.txt", "*.doc", ".rtf", "*.*" };
fd.setFilterExtensions(filterExt);
//打开文件对话框，返回选择的文件
String selected = fd.open();
System.out.println(selected);
```



图 10-8 文件对话框

10.5.5 目录对话框（DirectoryDialog）

目录对话框和文件对话框类似，用户可以通过目录对话框选择相应的目录，如图 10-9 所示。

新建目录对话框，用户可以设置对话框的初始路径、标题和提示信息等属性，最后返回选择的目录，如下代码片段所示。

```
//新建目录对话框
DirectoryDialog dlg = new DirectoryDialog(shell);
//设置初始路径
dlg.setFilterPath(text.getText());
//设置对话框标题
```

```
dlg.setText("SWT's DirectoryDialog");
//设置提示信息
dlg.setMessage("Select a directory");
//打开对话框，返回目录
String dir = dlg.open();
if (dir != null) {
  // Set the text box to the new selection
  text.setText(dir);
}
```



图 10-9　目录对话框

10.6　小　　结

对话框是 GUI 程序开发中经常用到的组件。本章介绍了常用的一些对话框，以及用户怎样自定义对话框，读者应该对对话框的功能和使用场景有所了解。另外，本章的对话框都能从本书源代码中找到出处。

第 11 章　SWT 的其他应用

在 SWT 中，不但提供了窗口组件，还提供了其他方面的一些功能，例如组件间的数据拖放等。

本章将介绍如下几个方面的内容。

- 拖放（Drag and drop）
- 线程（Thread）
- 剪贴板（Clipboard）
- 图形上下文（GC）
- 鼠标形状（Cursor）

11.1　拖　　放

鼠标拖放（Drag and drop）是 GUI 开发过程必备技巧。SWT 实现鼠标拖放方式介绍如下。

11.1.1　拖放的功能

在 SWT 的开发中，经常会涉及到鼠标拖放（Drag and drop）方面的应用，例如要开发一个文件浏览器，就要涉及到拖动文件，实现文件的拷贝移动，就要用到拖放的功能。

鼠标拖放涉及到两个操作，即鼠标的拖和放，这就需要实现源和目的两种类型的组件，以及拖放源和拖放目标。实现拖放的步骤如下。

（1）创建拖放源组件（即以哪一个组件作为拖放的起始组件），Control 类的所有子类都可以作为拖放源组件。

（2）创建拖放源，并把创建的拖放源组件作为拖放的起始组件，例如“DragSource ds = new DragSource(tree, DND.DROP_MOVE);”。

（3）设置拖放数据类型，例如拖放文本类型“ds.setTransfer(new Transfer[] { TextTransfer.getInstance() });”。

（4）为拖放源添加监听事件，例如“ds.addDragListener(new DragSourceAdapter());”。

（5）创建拖放目标组件，Control 类的所有子类都可以作为拖放目标组件。

（6）创建拖放目标，并把新建的组件作为拖放的目标组件，例如“DropTarget dt = new DropTarget(button, DND.DROP_MOVE);”。

（7）设置拖放数据类型，例如拖放文本类型“dt.setTransfer(new Transfer[] { TextTransfer.getInstance() });”。

（8）为拖放目标添加监听事件，例如“dt.addDropListener(new DropTargetAdapter());”。

11.1.2 拖放的实现

用户可以在SWT的组件之间拖放任意类型的数据。本节实例以文本类型的数据作为演示，代码如例程11-1所示。

例程11-1 SimpleDNDExample.java

```
package com.free.others;

import org.eclipse.swt.SWT;
import org.eclipse.swt.dnd.DND;
import org.eclipse.swt.dnd.DragSource;
import org.eclipse.swt.dnd.DragSourceAdapter;
import org.eclipse.swt.dnd.DragSourceEvent;
import org.eclipse.swt.dnd.DropTarget;
import org.eclipse.swt.dnd.DropTargetAdapter;
import org.eclipse.swt.dnd.DropTargetEvent;
import org.eclipse.swt.dnd.TextTransfer;
import org.eclipse.swt.dnd.Transfer;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Tree;
import org.eclipse.swt.widgets.TreeItem;

public class SimpleDNDExample {

  public static void main(String[] args) {
    Shell shell = new Shell();
    shell.setLayout(new FillLayout());

    //添加树组件
    final Tree tree = new Tree(shell, SWT.NONE);
    //添加树子节点
    TreeItem item1 = new TreeItem(tree, SWT.NONE);
    item1.setText("Item 1");
    TreeItem item2 = new TreeItem(tree, SWT.NONE);
    item2.setText("Item 2");
    TreeItem item3 = new TreeItem(tree, SWT.NONE);
    item3.setText("Item 3");
    TreeItem item4 = new TreeItem(tree, SWT.NONE);
    item4.setText("Item 4");

    //新建拖放的数据源
    DragSource ds = new DragSource(tree, DND.DROP_MOVE);
    //设置数据源可拖入的类型
    ds.setTransfer(new Transfer[] { TextTransfer.getInstance() });
    //添加拖动事件监听器
    ds.addDragListener(new DragSourceAdapter() {
```

```
        public void dragSetData(DragSourceEvent event) {
        //设置当前选定的第一个节点为拖动数据
          event.data = tree.getSelection()[0].getText();
        }
      });

      final Button button = new Button(shell, SWT.FLAT);
      button.setText("Button");
      button.setAlignment(SWT.CENTER);

      //新建拖放的目标
      DropTarget dt = new DropTarget(button, DND.DROP_MOVE);
      //设置可接收数据的类型
      dt.setTransfer(new Transfer[] { TextTransfer.getInstance() });
      //设置拖放事件监听器，监听鼠标放下事件
      dt.addDropListener(new DropTargetAdapter() {
        public void drop(DropTargetEvent event) {
          //设置按钮显示文本为数据源的文本
          button.setText((String) event.data);
        }
      });

      shell.pack();
      shell.open();
      Display display = Display.getDefault();
      while (!shell.isDisposed())
        if (!display.readAndDispatch())
          display.sleep();
      display.dispose();
    }

  }
```

程序中新建了拖放的数据源（Tree）和目标（Button），可以通过拖动 Tree 的节点信息给 Button 的显示文本赋值，程序运行效果如图 11-1 所示。

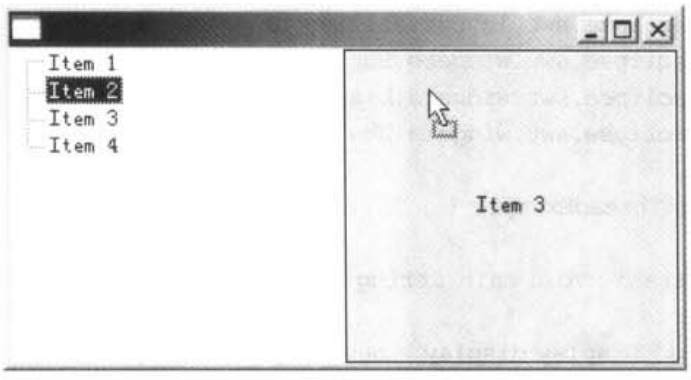


图 11-1　拖放的实现

11.2 线　　程

在多任务操作系统中，线程是比较重要的一个概念，本节将简要介绍线程的基本概念及通过 Display 操作线程的方式。

11.2.1 线程同步和异步

线程（Thread）是软件开发中经常会要用到的技术。在多线程的环境中，线程同步和线程异步是很重要的概念。线程异步是指多线程同时并发，多个线程同时运行时，每个线程的执行顺序是不确定的，通过线程异步执行，操作系统能够更好地利用时间片，从而获得更高的效率。线程同步是指多个线程同时存在时，线程的执行顺序是确定的，线程同步也是很有用的，例如，多个线程访问资源时，由于执行顺序确定，不会产生数据不同步的情况。

在 SWT 中，Display 通过 syncExec 和 asyncExec 提供了同步和异步的机制，从而实现同步和异步方式调用线程。

11.2.2 线程同步和异步的实现

下面将通过实例介绍线程的同步和异步。在本实例中，只是把两个按钮事件加入了 Display 的同步和异步方法，在具体的实例中，用户可以把 Display 的相应方法也放入线程中执行，代码如例程 11-2 所示。

例程 11-2　ThreadExample.java

```
package com.free.others;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class ThreadExample {

    public static void main(String[] args) {

        final Display display = new Display();
        Shell shell = new Shell(display);

        //新建一个线程
        final Runnable print = new Runnable() {
            public void run() {
```

```
            for (int i = 0; i < 100; i++){
                try {
                    Thread.sleep(100);
                    System.out.println("Print from thread: \t"
                            + Thread.currentThread().getName());
                } catch (Exception e) {
                    // TODO: handle exception
                }
            }
        }
    };

    shell.setText("Thread Example");
    shell.setSize(300, 100);
    shell.setLayout(new FillLayout());
    Button button = new Button(shell, SWT.CENTER);
    button.setText("asyncExec start");
    button.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            //线程异步执行
            System.out.println("start");
            display.asyncExec(print);
            System.out.println("stop");
        }
    });

    Button button1 = new Button(shell, SWT.CENTER);
    button1.setText("syncExec start");
    button1.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            //线程同步执行
            System.out.println("start");
            display.syncExec(print);
            System.out.println("stop");
        }
    });

    shell.open();
    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) {
            display.sleep();
        }
    }
    display.dispose();
  }
}
```

单击“syncExec start”按钮，程序将在最后打印出来“stop”文本，说明线程是同步方式执行（相对主线程来说），单击“asyncExec start”按钮，“stop”文本打印顺序不确定，说明线程是异步方式执行，程序运行效果如果 11-2 所示。

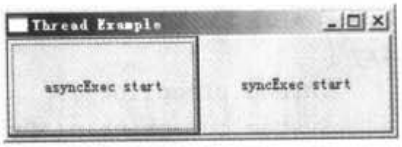


图 11-2　线程同步和异步的实现

11.3　剪　贴　板

在不同的操作系统中，剪贴板的实现方式不一样，SWT 中通过 Clipboard 对系统剪贴板进行了封装，具体使用如下。

11.3.1　剪贴板的功能

在 SWT 中提供了 Clipboard 对象对剪贴板进行封装，Clipboard 能把数据保存在剪贴板上，还能够获得剪贴板的数据。使用剪贴板的步骤如下。

（1）创建剪贴板对象，例如“Clipboard cb = new Clipboard(display);”。

（2）得到类型转换器实例，并设置数据到剪贴板中，例如“cb.setContents(data, TextTransfer.getInstance());”。

（3）得到类型转换器实例，并从剪贴板取出数据，例如“String string = (String) (cb.getContents(TextTransfer.getInstance()));”。

其中，TextTransfer.getInstance()是文本类型转换器的实例。因为 Java 是跨平台的语言，每种类型的操作系统对数据类型和剪贴板的定义都不一样，所以剪贴板中还要设置类型转换器。

11.3.2　剪贴板的实现

下面将通过实例演示如何使用剪贴板。该实例中定义了一个多行的文本框和与文本框关联的菜单，通过 Copy 菜单和 Paste 菜单实现剪贴板的拷贝和粘贴，代码如例程 11-3 所示。

例程 11-3　ClipboardExample.java

```
package com.free.others;

import org.eclipse.swt.SWT;
import org.eclipse.swt.dnd.Clipboard;
import org.eclipse.swt.dnd.TextTransfer;
import org.eclipse.swt.dnd.Transfer;
import org.eclipse.swt.dnd.TransferData;
import org.eclipse.swt.events.MenuAdapter;
import org.eclipse.swt.events.MenuEvent;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Menu;
```

```
import org.eclipse.swt.widgets.MenuItem;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

public class ClipboardExample {

  public static void main(String[] args) {
    Display display = new Display();
    //添加剪贴板对象
    final Clipboard cb = new Clipboard(display);
    Shell shell = new Shell(display);
    shell.setLayout(new FillLayout());
    final Text text = new Text(shell, SWT.BORDER | SWT.MULTI | SWT.WRAP);
    //添加菜单
    Menu menu = new Menu(shell, SWT.POP_UP);
    //添加 Copy 的菜单项
    final MenuItem copyItem = new MenuItem(menu, SWT.PUSH);
    //设置菜单项文字
    copyItem.setText("Copy");

    copyItem.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
        String selection = text.getSelectionText();
        if (selection.length() == 0)
          return;
        //获得选择的数据
        Object[] data = new Object[] { selection };
        //获得平台文本类型和 Java 文本类型的转换器
        Transfer[] types = new Transfer[] { TextTransfer.getInstance() };
        //设置剪贴板内容
        cb.setContents(data, types);
      }
    });
    //添加 Paste 的菜单项
    final MenuItem pasteItem = new MenuItem(menu, SWT.PUSH);
    pasteItem.setText("Paste");
    pasteItem.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent e) {
      //获得剪贴板的内容
        String string = (String) (cb.getContents(TextTransfer.getInstance()));
        //插入剪贴板的内容
        if (string != null)
          text.insert(string);
      }
    });
    menu.addMenuListener(new MenuAdapter() {
      public void menuShown(MenuEvent e) {
        //判断 Copy 菜单项是否可用
        String selection = text.getSelectionText();
        copyItem.setEnabled(selection.length() > 0);
        //判断 Paste 菜单项是否可用
```

```
            TransferData[] available = cb.getAvailableTypes();
            boolean enabled = false;
            for (int i = 0; i < available.length; i++) {
              if (TextTransfer.getInstance().isSupportedType(available[i])) {
                enabled = true;
                break;
              }
            }
            pasteItem.setEnabled(enabled);
          }
        });
        //在文本区域添加菜单
        text.setMenu(menu);

        shell.setSize(200, 200);
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch())
            display.sleep();
        }
        cb.dispose();
        display.dispose();
      }
    }
```

以上代码实现了右键菜单，可以通过右键菜单选择拷贝和粘贴对象到剪贴板，程序运行效果如图 11-3 所示。

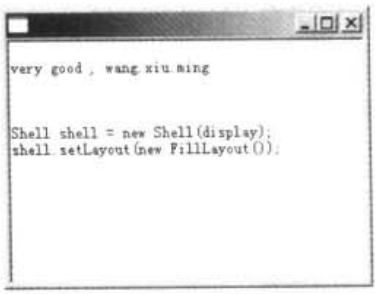


图 11-3　剪贴板的实现

11.4　图形上下文

图形上下文（GC）能在 Drawable 的对象上操作绘图区域，如果要自定义组件就要使用到 GC，具体介绍如下。

11.4.1　图形上下文的实现

在 SWT 中，GC（Graphics context）代表图形上下文，它能在 Drawable 的对象上操作绘图区域（Control 也实现了 Drawable 接口）。GC 中提供了很多方法，可以直接在 Drawable 对

象上画图，另外当使用完 GC 后，一定要调用 GC.dispose()释放操作系统分配的 GC 资源。

下面通过实例演示如何使用图形上下文，代码如例程 11-4 所示。

例程 11-4　GcExample.java

```
package com.free.others;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.PaintEvent;
import org.eclipse.swt.events.PaintListener;
import org.eclipse.swt.graphics.GC;
import org.eclipse.swt.widgets.Canvas;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class GcExample {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Drawing Example");

    final Canvas canvas = new Canvas(shell, SWT.NONE);
    canvas.setSize(150, 150);
    canvas.setLocation(20, 20);
    shell.open();
    shell.setSize(200, 220);

    //新建GC对象
    GC gc = new GC(canvas);
    //调用画图方法画矩形
    gc.drawRectangle(10, 10, 40, 45);
    gc.drawOval(65, 10, 30, 35);
    gc.drawLine(130, 10, 90, 80);
    gc.drawPolygon(new int[] { 20, 70, 45, 90, 70, 70 });
    gc.drawPolyline(new int[] { 10, 120, 70, 100, 100, 130, 130, 75 });
    //释放GC资源
    gc.dispose();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

上例通过 GC 的绘图方法对 Shell 组件绘图，实现了最原始的图形处理功能，程序运行效果如图 11-4 所示。

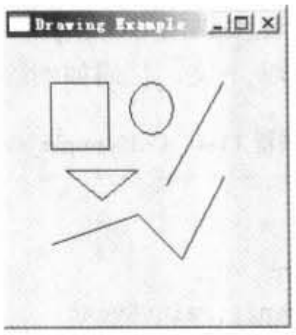


图 11-4　图形上下文实例

11.4.2　自定义组件

SWT 中提供的标准列表组件非常简单，只是提供字符串的选择。用户可能会根据一些特定的需求，自定义组件，这就要用到 GC 对组件的界面进行定制。自定义组件要考虑以下几个问题。

（1）自定义组件的绘制：通常用户需要自己对组件的形状或图案进行绘制。

（2）组件对键盘事件的响应：当焦点进入组件，用户进行键盘操作，通过键盘对组件进行控制时，需要让组件对用户的操作进行响应。例如在列表中，用户会通过上下箭头改变列表的选择项。

（3）组件对鼠标事件的响应：当用户用鼠标选中组件进行操作时，组件必须做出相应的反应。

（4）组件对焦点事件的响应：当界面焦点进入或移出组件，通常需要将组件绘制成得到或失去焦点的形状。例如，当焦点进入列表时，一般被选中的列表项会有虚框表示选中。

（5）响应【Tab】键：对于一个可操纵的组件，用户可以用【Tab】键将焦点移入或移出。

（6）响应滚动条事件：当组件有滚动条时，需要响应用户对滚动条的操作，完成对组件的绘制工作。

（7）提供事件监听机制：使用组件时通常需要监听组件中发生的一些事件，这样当事件发生时，组件能够进行相应处理。

（8）提供功能接口方便程序员访问：通常为方便程序员使用时获取组件中的信息或进行设置，组件需要提供一些接口。

组件开发最重要的就是组件的绘制，组件的绘制可以通过添加 PaintListener，在它的 paintControl 方法中进行，如下代码片段所示。

```
addPaintListener(new PaintListener() {
    public void paintControl(PaintEvent e) {
        GC gc = e.gc;
        Point size = getSize();
        int beginx = e.x;
        int beginy = (e.y / lineHeight) * lineHeight;
        int beginLine = (e.y - cy) / lineHeight;
        int endLine = beginLine + e.height / lineHeight + 1;
        if (endLine > getItemCount())
            endLine = getItemCount();
```

```
                for (int i = beginLine; i < endLine; i++) {
                    boolean selected = false;
                    if (i == rowSel)
                        selected = true;
                    onPaint(gc, i, cx, beginy + (i - beginLine) * lineHeight,
                            selected);
                }

            }
        });
```

另外，组件还要添加相应的监听器，当用户触发事件时响应用户的请求，如下代码片段如示。

```
        addControlListener(new ControlAdapter() {
            public void controlResized(ControlEvent event) {
                Point size = getSize();
                maxX = size.x * 3 / 2;
                maxY = colors.size() * lineHeight;
                resizeScrollBars();
            }
        });
        addFocusListener(new FocusListener() {
            public void focusGained(FocusEvent e) {
                ((Canvas) e.getSource()).redraw(cx, rowSel * lineHeight, maxX,
                        lineHeight, true);
            }

            public void focusLost(FocusEvent e) {
            ((Canvas) e.getSource()).redraw(cx, rowSel * lineHeight, maxX,
                            lineHeight, true);
                }
            });
            addTraverseListener(new TraverseListener() {
                public void keyTraversed(TraverseEvent e) {
                    if (e.detail == SWT.TRAVERSE_TAB_NEXT
                            || e.detail == SWT.TRAVERSE_TAB_PREVIOUS) {
                        e.doit = true;
                    }
                };

            });
```

读者可以从本书附带光盘中找到本节代码的完整实例 ColorList（自定义组件），测试代码类为 ColorListTest，程序运行效果如图 11-5 所示。



图 11-5　自定义组件

11.5　鼠标形状（Cursor）

SWT 组件中能够设置组件上鼠标的形状，用户可以通过 Cursor 创建鼠标的实例，然后调用组件的 setCursor 方法设置鼠标的实例。

新建鼠标形状的实例有多种方式，一般用户可以使用SWT提供的风格，例如“Cursor cursor = new Cursor(display, SWT.CURSOR_HAND);”创建手形状的鼠标光标，读者可以通过查找“SWT.CURSOR...”选取想要的鼠标风格。下面将通过实例演示如何从文件读取鼠标形状信息，代码如例程 11-5 所示。

例程 11-5　CursorExample.java

```
package com.free.others;

import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Cursor;
import org.eclipse.swt.graphics.ImageData;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import org.eclipse.swt.widgets.FileDialog;
import org.eclipse.swt.widgets.Listener;
import org.eclipse.swt.widgets.Shell;

public class CursorExample {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setSize(150, 150);
    final Cursor[] cursor = new Cursor[1];
    Button button = new Button(shell, SWT.PUSH);
    button.setText("Change cursor");
    Point size = button.computeSize(SWT.DEFAULT, SWT.DEFAULT);
    button.setSize(size);
```

```
        //添加按钮响应事件
        button.addListener(SWT.Selection, new Listener() {
          public void handleEvent(Event e) {
          //新建文件对话框
            FileDialog dialog = new FileDialog(shell);
            //设置文件对话框的过滤器
            dialog.setFilterExtensions(new String[] { "*.ico", "*.gif",
                "*.*" });
            String name = dialog.open();
            if (name == null)
              return;
            ImageData image = new ImageData(name);
            Cursor oldCursor = cursor[0];
            //创建鼠标
            cursor[0] = new Cursor(display, image, 0, 0);
            //设置 shell 上鼠标的形状
            shell.setCursor(cursor[0]);
            if (oldCursor != null)
              oldCursor.dispose();
          }
        });
        shell.open();
        while (!shell.isDisposed()) {
          if (!display.readAndDispatch())
            display.sleep();
        }
        if (cursor[0] != null)
          cursor[0].dispose();
        display.dispose();
      }
    }
```

上例从文件中读取图片的信息，并用此图片创建新的鼠标实例，程序运行效果如图 11-6 所示。图中的小飞机就是通过文件创建的鼠标图形。

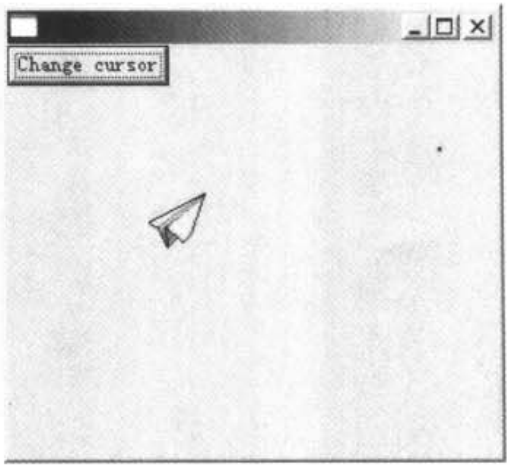


图 11-6　鼠标形状

11.6 小　　结

本章介绍了其他一些和 SWT 相关的应用，但 SWT 的功能不仅局限于此，而且 SWT 还是一个不断发展、不断完善，且开源的库，读者应该在掌握 SWT 基本知识的基础上不断深入研究。

最后提醒一句，阅读 SWT 的优秀代码是学习 SWT 的最好手段，希望读者在掌握 SWT 基本组件、事件和布局技术的基础上，不断探索 SWT/JFace 底层的实现，通过对它的研究，相信在不久的将来，读者会受益匪浅。

Eclipse 插件开发方法与实战

第 3 部分　Eclipse 插件开发篇

Eclipse 最有魅力的地方就是它的插件体系结构。在这个体系中重要的概念是扩展点（extension points）。扩展点就是在软件开发过程中暴露出来的接口。每一个插件都是在现有的扩展点上开发的，并可能还留有自己的扩展点，以便在这个插件上继续开发。

由于有了插件，Eclipse 系统的核心部分在启动时要完成的工作十分简单：启动平台的基础部分和查找系统的插件。整个 Eclipse 体系结构就像一个大拼图，可以不断地向上加插件，同时，在现有插件上还可以再加插件。

通过前面两个部分的学习，读者应该了解了 SWT/JFace，以及如何在 Eclipse 中进行 Java 开发，这些是插件开发的基础。本部分将会介绍 Eclipse 中常用插件扩展点的实现，结合 Java 开发，用户可以在 Eclipse 的基础上开发功能强大的插件。

第 12 章　插 件 入 门

Eclipse 平台是 IBM 向开发源码社区捐赠的开发框架，它之所以出名并不是因为 IBM 宣称投入开发的资金总数为 4 000 万美元，而是因为如此巨大的投入所带来的成果：一个成熟的、精心设计的、可扩展的体系结构。Eclipse 的价值是它为创建可扩展的集成开发环境提供了一个开放源码平台。这个平台允许任何人构建与环境和其他工具无缝集成的工具。工具与 Eclipse 无缝集成的关键是插件。除了小型的运行时内核之外，Eclipse 中的所有东西都是插件。从这个角度来讲，所有功能部件都是以同等的方式创建的。

本章将介绍如下几个方面的内容。

- 插件的基本概念
- 创建插件
- 插件清单文件
- 常用扩展点

12.1　什么是插件

Eclipse 理论上可以在插件的基础上无限扩展，如图 12-1 所示。

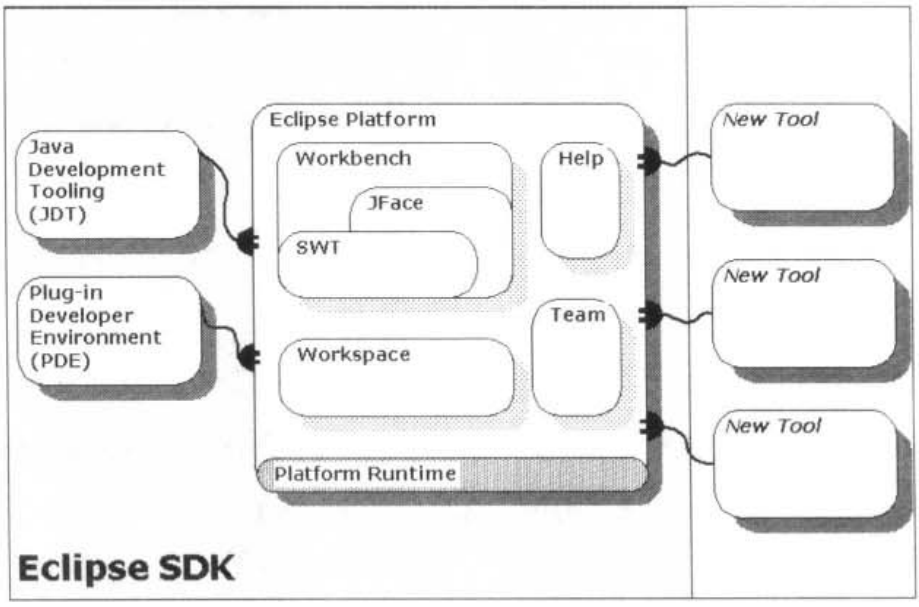


图 12-1　Eclipse 中的插件扩展机制

12.2 一个简单的插件

创建插件最简单的方法是使用 Eclipse 中专门为开发插件而设计的插件 PDE（Plug-in Development Environment）。PDE 和 Java Development Tooling（JDT）IDE 是 Eclipse 的标准扩展。PDE 提供了一些向导可以帮助创建插件。下面的“Hello，world”插件将通过 PDE 进行开发。

12.2.1 创建“Hello，world”插件

下面是创建一个简单插件的操作步骤。

（1）运行 Eclipse。

单击 Eclipse 的“File”→“New”→“Other”菜单项，在弹出的对话框中选择 Select 对话框左边的 Plug-in Development 向导。如图 12-2 所示，选择 Plug-in Project。

（2）单击“Next”按钮，弹出新建对话框，输入项目名称。此处使用了“com.test.helloworld”。单击“Next”按钮后弹出新建对话框页，如图 12-3 所示，插件标识就与项目名称相同。使用项目名称作为插件标识可以将该插件与另一个插件的名称发生冲突的机会减到最小。

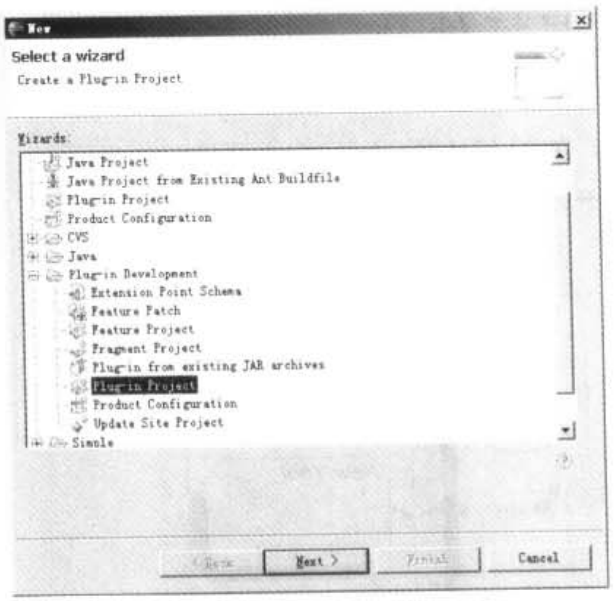


图 12-2 新建插件对话框

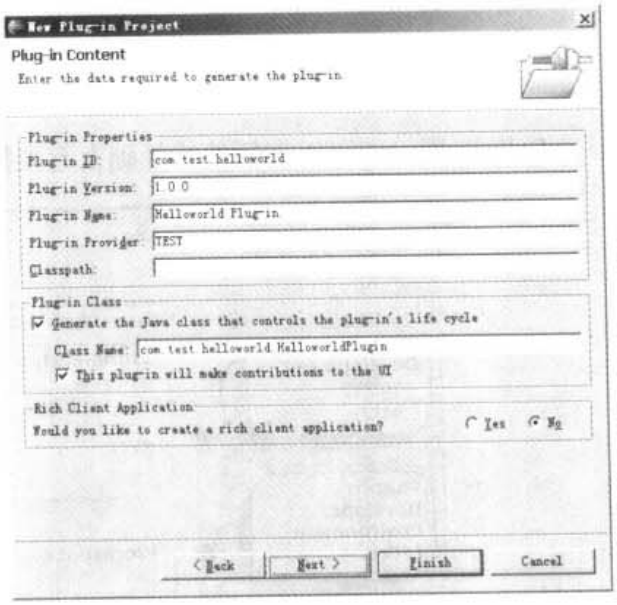


图 12-3 新建对话框向导

（3）单击“Next”按钮，选择“Hello, world”，如图 12-4 所示。

（4）单击“Finish”按钮，就可以创建用户想要建立的插件。



图 12-4　新建插件向导

12.2.2　调试“Hello，world”插件

通过 PDE 不但能创建插件，还能够调试插件。在 Eclipse 中调试插件的步骤如下。

（1）单击“Run”→“Debug”菜单项。

（2）在弹出的对话框窗口中用鼠标右键单击“Eclipse Application”选项。

（3）单击“New”菜单项（或双击 Eclipse Application 树节点），创建调试插件配置参数对话框，并通过对话框设置插件的调试参数，如图 12-5 所示。

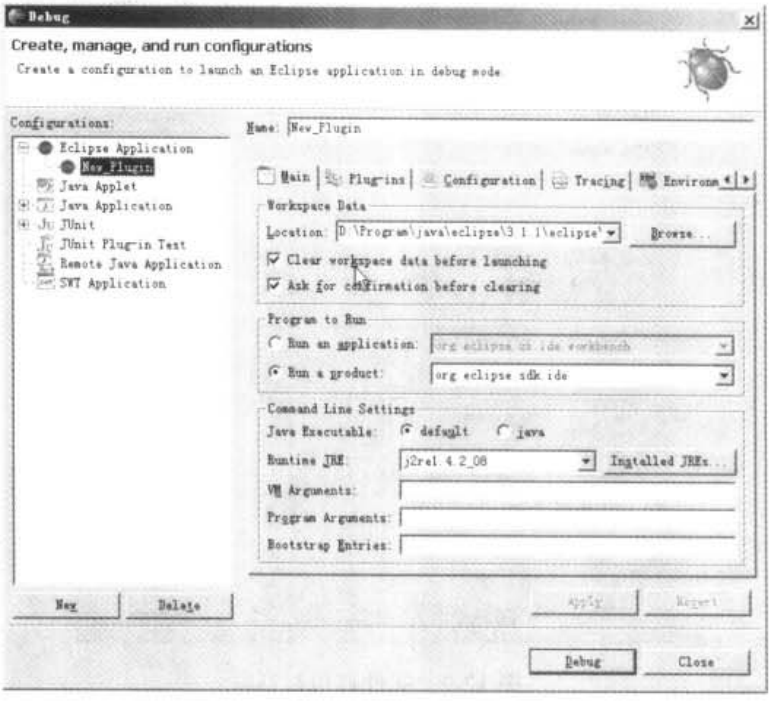


图 12-5　Debug 环境参数设置

（4）单击“Debug”按钮。

现在已经启动了一个调试的 Eclipse，可以看到图 12-6 显示的调试窗口。调试窗口多了一个“Sample Menu”菜单项，工具栏多了一个按钮，单击菜单或按钮将会弹出 “Hello，world”对话框。



图 12-6 “Hello，world”插件效果图

用户可以在程序中可执行到的位置设置断点，就可以和调试 Java 程序一样调试 Eclipse 插件了。

> **注意** 如果选择 Clear workspace data before lauching，表示每次调试插件时是否提示清空运行时刻的 Workspace。可以选择“是”，表示重建运行时刻的 Workspace，相当于把修改的插件再重新载入一次。

12.2.3 插件打包

这一步是把用户开发的插件打包，供别人使用。具体步骤如下。

（1）选择“Hello，world”插件，单击鼠标右键，选择“Export”菜单，弹出图 12-7 所示的对话框。

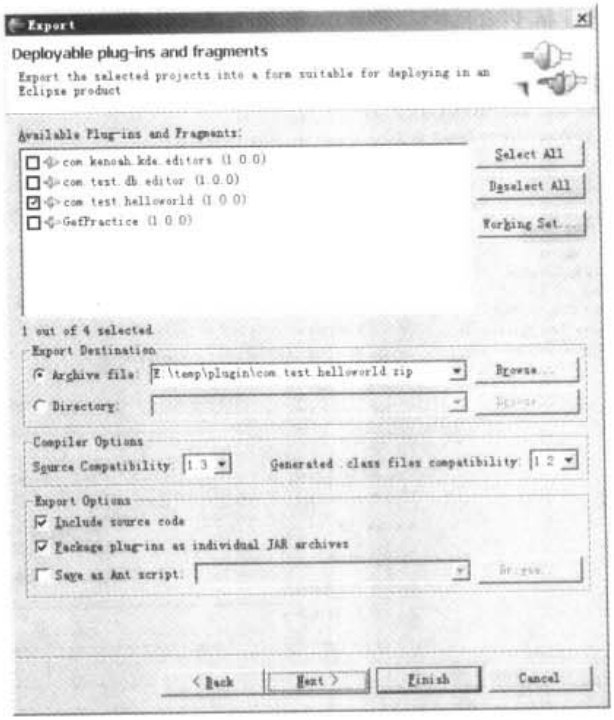


图 12-7 插件打包对话框

（2）输入用户想要打包的文件名。如果想把源文件也打包，可以选择“Include source code”

复选框。现在“Hello，world”插件就已经成功打包了。

（3）把“Hello，world”插件解压缩拷贝到用户的 Eclipse 的 Plugins 目录下面。运行 Eclipse，就可以在运行环境下看到“Hello，world”插件的身影了。

> **注意**　不要把“Hello，World”插件拷贝到开发环境的 Eclipse 下面，否则插件可能不会被注册。因为开发环境的 Workspace 中有相同 ID 的插件，这样会产生冲突。要养成好习惯，把开发环境和部署环境分开。

12.3　插件清单文件介绍

Plugin.xml 文件是插件开发中最常用到的文件，它的主要功能是通过 XML 文件格式描述插件的具体内容。当 Eclipse 启动后，它将会找到所有有效的 plugin.xml 文件。当 Eclipse 第一次启动某一个插件时，它会从 plugin.xml 文件读取插件的相关信息，并初始化插件。

plugin.xml 文件编辑器窗口，如图 12-8 所示。

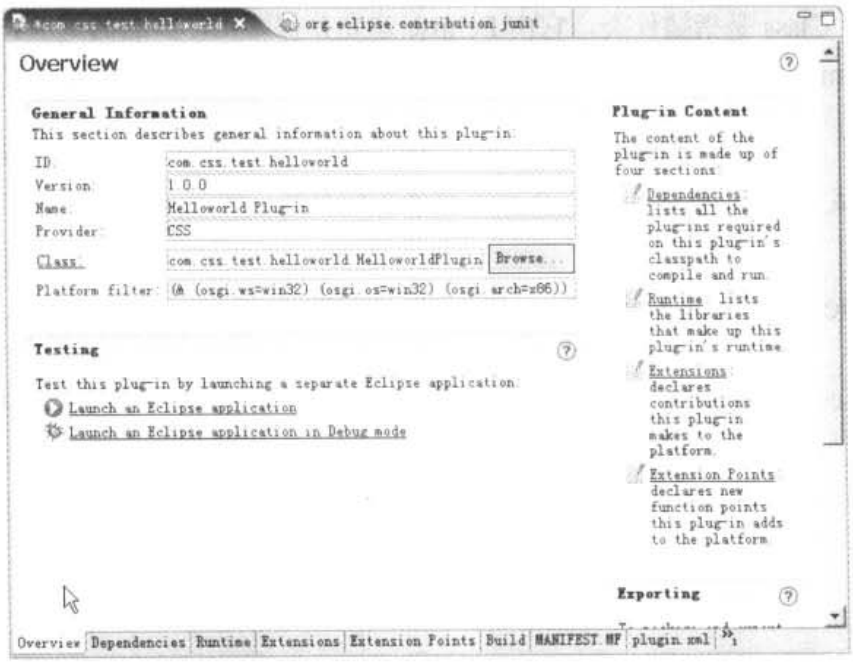


图 12-8　清单文件编辑窗口

另外，如果新建一个插件时选择创建 Manifest 文件，则 Eclipse 会把基本信息放入 Manifest.MF 文件中保存（和 OSGI 相关的信息），否则会保存在 Plugin.xml 文件中。Eclipse 3.1 版本以后会默认创建 Manifest.MF 文件。

Plugin.xml 文件分成如下几个部分。

- Overview：描述了插件的基本信息。
- Dependencies：描述了插件的依赖关系。
- Runtime：指明运行时的 ClassPath。
- Extensions：指明插件实现的扩展点。
- Extension Points：指明插件提供的扩展点。

12.3.1 OverView：描述了插件的基本信息

OverView 描述了插件的基本信息，如图 12-9 所示。

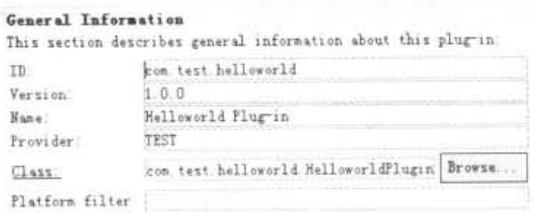


图 12-9 OverView 页面

- ID：ID 是插件的 ID 号，在 Eclipse 中是一个具体标识。
- Version：Version 指明用户所开发的插件的版本号。
- Name：指用户所开发的插件的名称。
- Provider：开发者。
- Class：Class 是指插件类，它由 Eclipse 建立并初始化，后面的章节将会详细介绍。
- Platform Filter：指定平台相关的一些信息，一般来说用户不需要设定。



ID 号在 Eclipse 中是一个全局的标识。很多初学者常常会对不同的插件命名成相同的 ID 号，这样会导致插件不能加载。

12.3.2 Dependencies：描述了插件的依赖关系

Dependencies 页面描述了插件的依赖关系，如图 12-10 所示，“Hello，World”插件依赖 org.eclipse.ui 和 org.eclipse.core.runtime 插件。



图 12-10 Dependencies 页面

Dependencies 是一个比较重要的概念。每一个插件有独立的 ClassPath，Dependencies 会把依赖的插件的 ClassPath 加入到当前的 ClassPath 中。例如在 SWT 开发的章节里面，笔者采用创建插件的方式建立一个插件项目，插件项目会把当前插件所依赖的包加入当前项目的查找路径中，其中就包括了 SWT/JFace 实现的 Jar 包。

12.3.3　Runtime：指明了运行时的 ClassPath

Runtime：指明了运行时的 ClassPath。如图 12-11 所示，library.jar 为此插件运行时依赖的包。

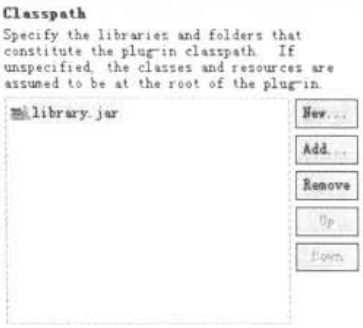


图 12-11　Runtime 页面

Runtime 也是一个比较重要的概念，它指明了运行时的环境。Eclipse 中插件不会引用到系统的 ClassPath，每一个插件都有独立的 ClassPath，这一个 ClassPath 由 Runtime 和 Dependencies 共同指明。

另外，很多人在开发插件时，没有通过 Runtime 指明 ClassPath，而是直接添加 Jar 包，这样，在编译期间不会出错，但是在插件运行时会提示类找不到，就是这个问题。

12.3.4　Extensions：指明插件实现的扩展点

Extensions：指明插件实现的扩展点。例如插件要实现“org.eclipse.ui.actionSets”扩展点，如图 12-12 所示。



图 12-12　Extensions 页面

Extensions 是用户在开发一个插件中用到的扩展点，也是 Eclipse 开发的精髓所在。后面章节的主要工作就是围绕扩展点展开的。下面为扩展点在 Plugin.xml 文件中定义的片段。

```
<!-- 实现的扩展点 -->
<extension
      point="org.eclipse.ui.actionSets">
   <actionSet
      label="Sample Action Set"
      visible="true"
      id="com.test.helloworld.actionSet">
      <menu
```

```
            label="Sample &amp;Menu"
            id="sampleMenu">
          <separator
              name="sampleGroup">
          </separator>
        </menu>
        <action
            label="&amp;Sample Action"
            icon="icons/sample.gif"
            <!-- 扩展点的实现类 -->
            class="com.test.helloworld.actions.SampleAction"
            tooltip="Hello, Eclipse world"
            menubarPath="sampleMenu/sampleGroup"
            toolbarPath="sampleGroup"
            id="com.test.helloworld.actions.SampleAction">
        </action>
      </actionSet>
    </extension>
```

12.3.5 Extension Points：指明了用户插件提供的扩展点

Extension Points：指明了用户插件提供的扩展点。如图 12-13 所示，此处为 UI 插件所提供的扩展点。



图 12-13 插件提供扩展点

Eclipse 的强大功能就是提供插件的扩展机制，用户可以实现他人提供的扩展点，也可以通过“Extension Points”为他人提供扩展点。

另外，实现的扩展点和提供的扩展点是两个不同的概念。实现的扩展点是利用其他插件提供的扩展点，实现用户想要完成的功能。提供的扩展点是为其他插件提供扩展所需要的接口。

12.4 常用扩展点的描述

在开发 Eclipse 中常用的扩展点很多，这里只简要介绍几个扩展点，后面的章节将会具体介绍扩展点的实现。

12.4.1 org.eclipse.ui.editors：代表一个编辑器窗口

Editors 扩展点是用户在开发中最常用的扩展点之一，它代表一个编辑器窗口。通过扩展“org.eclipse.ui.editors”，用户能在 Eclipse 中实现自己的编辑器，并能通过 Eclipse 的强大功能对用户的编辑器进行扩展。下面是编辑器扩展点在 pulgin.xml 文件中的定义，后面在“Hello, World”中将会深入介绍 org.eclipse.ui.editors 扩展点的用法。

如下代码片段为 Editor 扩展点在 Plugin.xml 文件中的定义。

```
<extension
   point = "org.eclipse.ui.editors">
    <editor
      id = "org.eclipse.ui.examples.readmetool.ReadmeEditor"
      name="%Editors.ReadmeEditor"
          icon="icons/obj16/editor.png"
          class="org.eclipse.ui.examples.readmetool.ReadmeEditor"
      extensions="readme"
        contributorClass="org.eclipse.ui.examples.readmetool.ReadmeEditorAct
ionBarContributor">
    </editor>
</extension>
```

“org.eclipse.ui.editors”扩展点用来将新的编辑器添加至工作台。编辑器是工作台页面内的可视组件。它通常用来编辑或浏览文档或输入对象。要打开编辑器，用户通常将对 IFile 调用“打开”。当执行此操作时，将查阅工作台注册表以便为文件类型确定适当的编辑器，然后创建编辑器类型的新实例。实际结果取决于编辑器的类型。工作台能够创建内部编辑器（它已紧密集成到工作台中）和外部编辑器（它是在单独的框架窗口中启动的）。这些编辑器之间还具有各种级别的集成。

在内部编辑器的情况下，可实现工作台窗口与编辑器部件之间的紧密集成。工作台菜单和工具栏预装入了许多公共操作（如剪切、复制和粘贴）。活动的部件（视图或编辑器）应会提供这些操作的实现。内部编辑器还可定义出现在工作台窗口中的新操作。仅当编辑器活动时，这些操作才会出现。

工作台与外部编辑器之间的集成则更为松散一些。在这种情况下，工作台可以启动编辑器，但自此以后，除了通过文件系统之外它再没有任何办法确定外部编辑器的状态或与它合作。

12.4.2 org.eclipse.ui.views：代表一个视图窗口

Views 扩展点能实现 Eclipse 中的视图，Eclipse 主要的功能部分是由编辑器和视图构成的。在 Eclipse 中，同一时间只能显示一个编辑器，但是能显示多个视图。一般在应用中，通常会

实现多个视图窗口，例如在下面的 plugin.xml 文件中就定义了 5 个视图，这些视图各自完成彼此的功能。在 Eclipse 中除了编辑窗口，其他的窗口都可以看成是视图的体现。

如下代码片段为 View 扩展点在 Plugin.xml 文件中的定义。

```
<extension
      point="org.eclipse.ui.views">
   <category
       name="%view.category"
       id="com.kenoah.kde.ui"/>
   <view
       class="com.kenoah.kde.ui.views.ListTableView"
       icon="imgs/ListTag.gif"
       category="com.kenoah.kde.ui"
       name="%view.listViewer"
       id="com.kenoah.kde.ui.views.ListTableView"/>

   <view
       icon="imgs/DynaTag.gif"
       class="com.kenoah.kde.ui.views.DynaPartViewer"
       category="com.kenoah.kde.ui"
       name="%view.dynPartViewer"
       id="com.kenoah.kde.ui.views.DynaPartViewer"/>
   <view
       icon="imgs/AppTag.gif"
       class="com.kenoah.kde.ui.views.AppViewer"
       category="com.kenoah.kde.ui"
       name="%view.appViewer"
       id="com.kenoah.kde.ui.views.AppViewer"/>
   <view
       icon="imgs/ApplicationResource.gif"
       class="com.kenoah.kde.ui.views.KenoahTreeViewer"
       category="com.kenoah.kde.ui"
       name="%view.KenoahTreeViewer"
       id="com.kenoah.kde.ui.views.KenoahTreeViewer"/>
    <view
       icon="imgs/TreeView.gif"
       class="com.kenoah.kde.ui.views.LookupTreeViewer"
       category="com.kenoah.kde.ui"
       name="%view.LookupTreeViewer"
       id="com.kenoah.kde.ui.views.LookupTreeViewer"/>

</extension>
```

“org.eclipse.ui.views”扩展点用来为工作台定义更多视图。视图是工作台页面内的可视组件。通常用来浏览信息的层次结构（如工作空间）、打开编辑器或显示活动编辑器的属性。用户可以使用“视图”菜单选择可见视图，或者从视图本地标题栏中关闭视图。

视图是可以浏览信息层次结构或显示对象属性的工作台部件。在一个工作台页面中，只能打开任何给定视图的一个实例。当用户在视图中进行选择或其他更改时，这些更改会立即显示在工作台中。提供视图通常是为了支持相应的编辑器。

12.4.3 org.eclipse.ui.perspectives：代表一个扩展视角

Perspectives 扩展点实现一个视角，视角直观来讲就是用户在某一时间能够看到的所有视图和编辑器的集合。在 Eclipse 中可以看到很多的视角，例如 Java 视角、Resource 视角、Debug 视角，用户可以通过不同的视角完成不同的任务。

视角不包含任何的编辑器和视图的实例，只包含这些编辑器和视图的引用。如果相应的视图没有打开，当你切换到选定的视角时，视角会自动打开此视图。

如下代码片段为 Perspectives 在 Plugin.xml 文件中的定义。

```
<extension
      point="org.eclipse.ui.perspectives">
   <perspective
       name="%Perspective.resourcePerspective"
       icon="icons/full/cview16/resource_persp.png"
       class="org.eclipse.ui.internal.ResourcePerspective"
       id="org.eclipse.ui.resourcePerspective">
   </perspective>
</extension>
```

此扩展点用来将视角添加至工作台。视角用来定义工作台布局和可视操作集。用户可通过调用“Window”→“Open perspective”菜单来选择视角。视角在工作台窗口内部提供附加组织层。当用户在任务之间移动时，可以在视角之间进行切换。视角定义视图集合、视图布局和用户首次打开视角时应使用的可视操作集。

12.5 小　　结

本章简要介绍了插件的基本概念，接着利用 Eclipse 提供的 PDE 向导生成了一个插件的基本应用。通过打包、发布，演示了构建一个插件的基本流程。

通过本章的学习，读者应该对插件的开发有个感性的认识，并能通过 PDE 创建插件项目，常用的扩展点比较多，后面的章节将会重点介绍常用扩展点的实现。

第 13 章　视图扩展点的实现

在 Eclipse 插件的开发中，视图插件的开发是比较重要的一个部分。视图是工作台页面内的可视组件，用户可以通过视图，扩展 Eclipse 原有的功能。

本章主要从以下几个部分进行介绍。

- Views 扩展点的介绍
- “Hello，world”视图
- “Tree viewer”视图
- “Image viewer”视图
- Debug 视图的实现

13.1　Views 扩展点的介绍

视图是工作台页面内的可视组件，通常用来浏览信息的层次结构（如工作空间）、打开编辑器或显示活动编辑器的属性。用户可以通过“Window”→“Show view”菜单显示视图，也可以从视图局部标题栏关闭视图。

13.1.1　视图分类

在 Eclipse 中用户可以在同一时间打开多个视图（编辑器在同一时间只能打开一个），每个视图能够有自己的位置布局信息，Eclipse 会根据上一次视图布局的信息初始化视图。

Eclipse 视图一般来说有两种表现形式，一种是用户通过视图扩展点“org.eclipse.ui.views”实现，另一种是 Eclipse 实现的视图，用户可以通过实现 Eclipse 的相应接口实现，如 Outline 视图和 Properties 视图，如图 13-1 所示。

在 Eclipse 中，用户可以按自己的要求建立自己的视图，即扩展 Eclipse 的视图扩展点。也能实现 Eclipse 的接口，通过 Eclipse 的 Adapter 机制实现 Eclipse 的扩展。其中，Outline 和 Properties 视图也是通过这种方式实现的，在后面的章节会介绍 Outline 视图和 Properties 视图的实现。

当用户定义了自己的视图后能通过“Window”→“Show view”菜单打开自定义的视图。如图 13-2 所示。

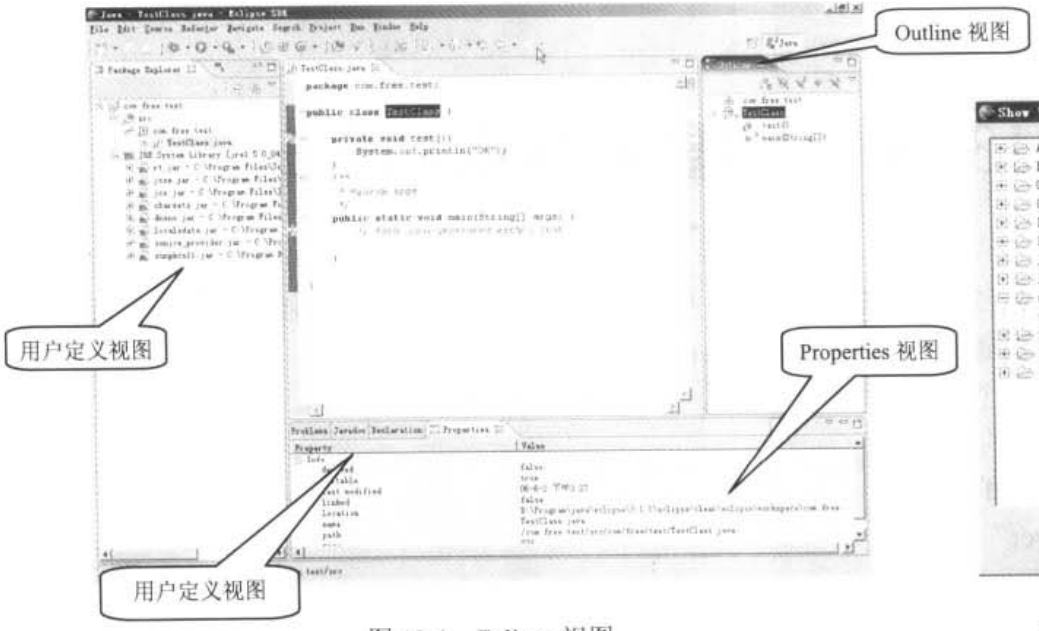


图 13-1　Eclipse 视图



图 13-2　选择视图

13.1.2　常用视图可实现的功能

Eclipse 的视图就功能来说一般有两种形式，一种是功能性视图，例如 Java 的 Default 视图，default 负责用树形目录组织 Java 的项目结构，并且能通过双击节点在编辑器中打开相应的资源。另一种视图是辅助型的视图，例如 Console 负责控制台信息的显示。

不管是哪种类型的视图，它们的实现原理是一样的。在 Eclipse 中视图可以实现的功能有如下一些。

- 右键菜单：可以通过视图中的树、表弹出的菜单完成相应的功能。
- 工具栏：可以定制视图的工具栏，完成当前视图的功能。
- 表格：视图中可以加入表格以显示相应的表结构信息。
- 树：视图中可以加入树结构以显示相应的树结构信息。
- Log 输出：视图中可以直接输出信息。



任何 SWT 的组件都可以作为视图的一部分，用户可以按自己的需求定制自己的视图。

13.2　“Hello，world”视图

本节目的在于让读者能通过一个简单的例子对视图有所了解，通过本节读者应该能够建立属于自己的视图。

13.2.1　创建“Hello，world”视图插件

通过 Eclipse 的向导建立一个插件项目，步骤如下。

（1）打开新建向导。

单击 Eclipse 的“File”→“New”→“Other”菜单项，在弹出的对话框中选择 Select 对话框左边的 Plug-in Development 向导。如图 13-3 所示，选择 Plug-in Project。

（2）建立一个空的插件项目。

① 输入插件工程的名字“com.free.view.helloworld”。

② 输入插件的 ID“com.free.view.helloworld”。

③ 输入插件的 Name“helloworld Plug-in”。

④ 不选择使用模板建立插件项目，如图 13-4 所示。

⑤ 单击“Finish”按钮，完成项目的建立。



图 13-3　新建插件对话框



图 13-4　选择插件模板对话框

通过这种方式建立了一个空的插件项目，要让此插件成为一个视图的插件，还要实现相应的扩展点。



读者可以通过选择插件模板的方式建立一个想要的插件，在此只是为了让读者了解如何在一个空白的插件中加入读者期望的扩展点。在后面的章节，笔者会介绍用插件模板建立插件。

13.2.2　“Hello，world”插件扩展点

为了实现视图插件，就应该实现 Eclipse 提供的视图扩展点，具体步骤如下。

（1）打开插件描述文件的 Extensions 页。

新建一个插件后，会自动打开插件清单文件编辑器，读者也可以通过 META-INF 目录下的 MENIFEST.MF 文件打开插件清单文件编辑器。

（2）单击“Add”按钮。

（3）选择“org.eclipse.ui.views”扩展点。

如图 13-5 所示，选择“org.eclipse.ui.views”扩展点，单击“Finish”按钮。读者也可以通

过向导选择模板生成相应的扩展点清单文件，在此笔者采用手动方式生成插件清单文件。

（4）新建一个视图分类节点（category）。

命名视图分类节点，如图 13-6 所示。

- id：“com.free.view.categoryall”
- name：“free view”

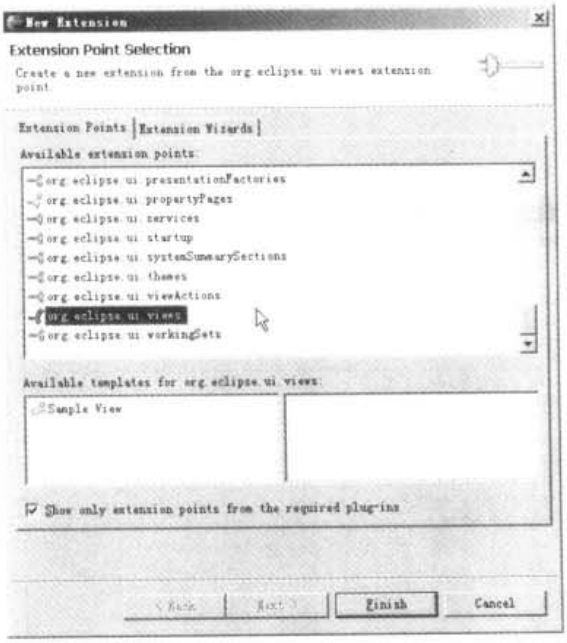


图 13-5　选择扩展点

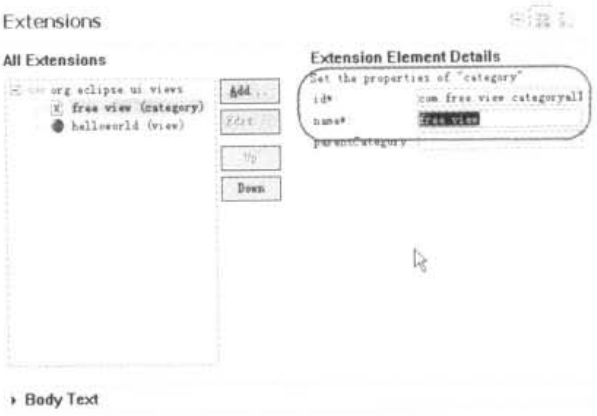


图 13-6　视图分类节点

Eclipse 采用视图分类节点的 ID 号来组织视图在“Show View”窗口的显示，在视图的扩展点中也可以不实现视图分类节点。

（5）新建视图节点（view）。

命名视图节点，如图 13-7 所示。

- id：“com.free.view.helloworld.view”
- name：“helloworld”
- class：“com.free.view.helloworld.HelloWorldView”
- category：“com.free.view.categoryall”
- icon：“icons/sample.gif”

视图节点的解释。

- id 为视图的编号，用户可以通过此编号在程序中访问该视图。
- class 为视图的实现类，当用户选择打开视图时，Eclipse 会自动调用此类初始化并维护视图。
- category 表示此视图属于哪一个分类，例如，用户可以输入“org.eclipse.ant.ui.views”，表示用户希望把自己的视图在“Show View”窗口的 ANT 分类节点显示。
- icon 为视图的图标。

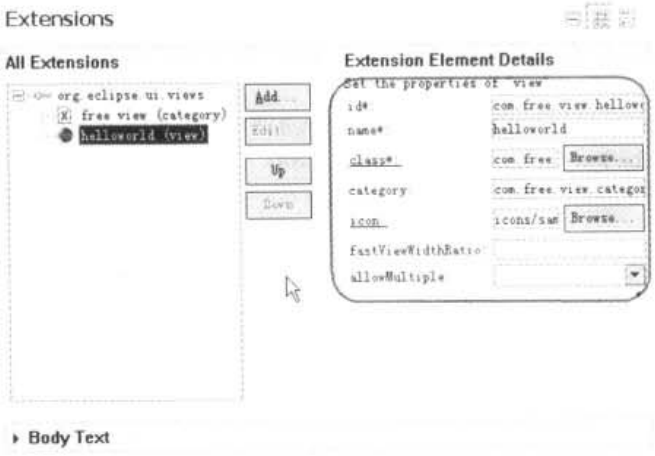


图 13-7　视图节点

“Hello，world”视图的 plugin.xml 文件，如例程 13-1 所示。

例程 13-1　plugin.xml

```
<plugin>
  <extension
      point="org.eclipse.ui.views">
    <category
        id="com.free.view.categoryall"
        name="free view"/>
    <view
        category="com.free.view.categoryall"
        class="com.free.view.helloworld.HelloWorldView"
        icon="icons/sample.gif"
        id="com.free.view.helloworld.view"
        name="helloworld"/>
  </extension>

</plugin>
```

13.2.3　“org.eclipse.ui.views”扩展点的实现类“HelloWorldView”

当用户选择视图后，Eclipse 会自动调用视图节点 Class 所指向的类（即扩展点实现类）对视图进行初始化。

用户定义的视图继承于“org.eclipse.ui.part.ViewPart”，且必须实现 createPartControl(Composite parent)方法和 setFocus()方法，如例程 13-2 所示。

例程 13-2　HelloWorldView.java

```
package com.free.view.helloworld;

import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.part.ViewPart;
```

```
public class HelloWorldView extends ViewPart {

    public void createPartControl(Composite parent) {
        Label label = new Label(parent, SWT.LEFT);
        label.setText("Hello, world!");
    }

    public void setFocus() {
        System.out.println("Hello, world!focus");
    }

}
```

当视图第一次运行时，Eclipse 会调用 createPartControl 方法构造视图，parent 参数为 SWT 的 Composite 组件，在此例子中只在 parent 的组件中加入了一个“Hello, world!”标签。另外，当视图被激活时，Eclipse 会调用 setFocus 方法。

13.2.4 运行“Hello，world”视图

“Hello，world”插件到此就已经完成，现在可以调试插件了。

（1）运行插件。

（2）打开运行时工作台的“show view”。

在“Show View”窗口中可以看到前面建立的分类和视图节点，如图 13-8 所示。

（3）运行“Hello，world”视图。

双击“helloworld”节点，打开视图窗口，如图 13-9 所示。

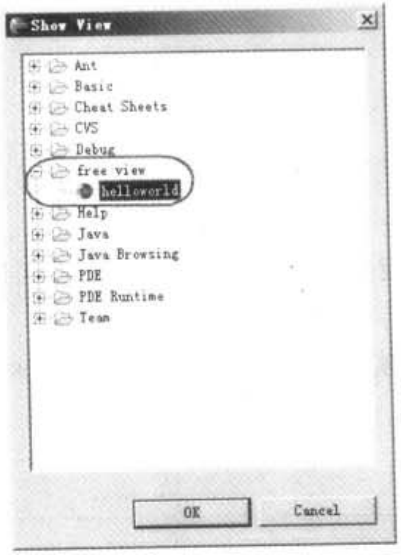


图 13-8 “Show View”窗口

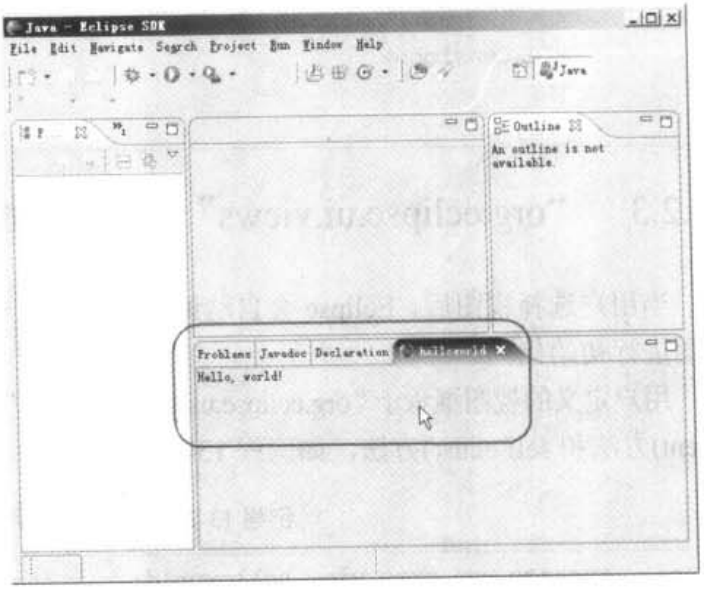


图 13-9 “Hello，world”视图

在宿主工作台的 Console 视图中能看到“Hello, world!focus”的输出信息。

13.3　“Tree viewer”视图

上一节中，视图窗口包含一个标签组件，用来显示“Hello，world”。本节将在视图中加入树组件和视图中常用的菜单和工具栏功能。

13.3.1　创建“Tree viewer”视图插件

创建“Tree viewer”视图插件项目，步骤如下。

（1）打开新建向导。

单击 Eclipse 的“File”→“New”→“Other”菜单项，在弹出的对话框中选择 Select 对话框左边的 Plug-in Development 向导。

（2）建立一个空的插件项目。

① 输入插件工程的名字为“com.free.view. treeview”。

② 输入插件的 ID 为“com.free.view.treeview”。

③ 输入插件的 Name 为“helloworld Plug-in”。

④ 选择使用模板建立插件项目，如图 13-10 所示。

⑤ 单击“Next”按钮，选择“Tree viewer”，如图 13-11 所示。

⑥ 单击“Finish”按钮，完成项目的建立。

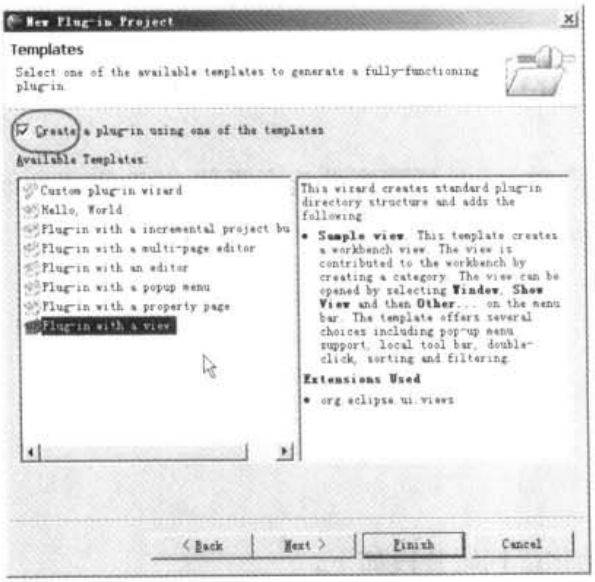


图 13-10　选择插件模板对话框



图 13-11　选择视图类型

通过这种方式建立了一个包含 JFace 的“Tree viewer”组件的插件项目，此项目已经具备了视图插件常用的功能。

13.3.2 “Tree viewer”插件扩展点

此插件和上一节的“Hello，world”插件类似，也只实现了一个视图分类节点和视图节点。如例程 13-3 所示。

例程 13-3 plugin.xml

```
<plugin>

  <extension
      point="org.eclipse.ui.views">
    <category
        name="tree view"
        id="com.free.view.categoryall">
    </category>
    <view
        name="Tree View"
        icon="icons/tree.GIF"
        category="com.free.view.categoryall"
        class="com.free.view.treeview.views.TreeViewTest"
        id="com.free.view.treeview.views.TreeViewTest">
    </view>
  </extension>

</plugin>
```

13.3.3 “org.eclipse.ui.views”扩展点的实现类“TreeViewTest”

在“Tree viewer”视图中要完成的功能有：树、菜单、工具栏和事件响应事件。“Tree viewer”视图的实现类为“TreeViewTest”，通过实现类能实现这些功能。

实现类中要完成如下的功能。

（1）初始化树。

（2）完成菜单和工具栏响应的 Action。

（3）完成上下文菜单。

（4）完成工具栏。

（5）完成双击树节点的响应方式。

在 createPartControl 方法中能对视图进行初始化，代码片段如下。

```
public void createPartControl(Composite parent) {
        //初始化树
        viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
        drillDownAdapter = new DrillDownAdapter(viewer);
        viewer.setContentProvider(new ViewContentProvider());
        viewer.setLabelProvider(new ViewLabelProvider());
        viewer.setSorter(new NameSorter());
        viewer.setInput(getViewSite());
          //完成菜单和工具栏响应的Action
```

```
        makeActions();
      //完成上下文菜单
        hookContextMenu();
      //完成双击树节点的响应方式
        hookDoubleClickAction();
      //完成工具栏
        contributeToActionBars();
    }
```

通过实现类，用户能加入自己感兴趣的功能。“Tree viewer”视图中实现类的完整代码如例程 13-4 所示。

例程 13-4　TreeViewTest.java

```
package com.free.view.treeview.views;

import java.util.ArrayList;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.Image;
import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.*;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.SWT;
import org.eclipse.core.runtime.IAdaptable;

public class TreeViewTest extends ViewPart {
    private TreeViewer viewer;
    private DrillDownAdapter drillDownAdapter;
    private Action action1;
    private Action action2;
    private Action doubleClickAction;
    //创建树节点
    class TreeObject implements IAdaptable {
        private String name;
        private TreeParent parent;

        public TreeObject(String name) {
            this.name = name;
        }
        public String getName() {
            return name;
        }
        public void setParent(TreeParent parent) {
            this.parent = parent;
        }
        public TreeParent getParent() {
            return parent;
        }
```

```
    public String toString() {
        return getName();
    }
    public Object getAdapter(Class key) {
        return null;
    }
}

class TreeParent extends TreeObject {
    private ArrayList children;
    public TreeParent(String name) {
        super(name);
        children = new ArrayList();
    }
    public void addChild(TreeObject child) {
        children.add(child);
        child.setParent(this);
    }
    public void removeChild(TreeObject child) {
        children.remove(child);
        child.setParent(null);
    }
    public TreeObject [] getChildren() {
        return (TreeObject [])children.toArray(new TreeObject[children.size()]);
    }
    public boolean hasChildren() {
        return children.size()>0;
    }
}
//实现内容提供器
class ViewContentProvider implements IStructuredContentProvider,
                                     ITreeContentProvider {
    private TreeParent invisibleRoot;

    public void inputChanged(Viewer v, Object oldInput, Object newInput) {
    }
    public void dispose() {
    }
    public Object[] getElements(Object parent) {
        if (parent.equals(getViewSite())) {
            if (invisibleRoot==null) initialize();
            return getChildren(invisibleRoot);
        }
        return getChildren(parent);
    }
    public Object getParent(Object child) {
        if (child instanceof TreeObject) {
            return ((TreeObject)child).getParent();
        }
        return null;
    }
```

```
    public Object [] getChildren(Object parent) {
        if (parent instanceof TreeParent) {
            return ((TreeParent)parent).getChildren();
        }
        return new Object[0];
    }
    public boolean hasChildren(Object parent) {
        if (parent instanceof TreeParent)
            return ((TreeParent)parent).hasChildren();
        return false;
    }
   //初始化树节点
    private void initialize() {
        TreeObject to1 = new TreeObject("Leat 1");
        TreeObject to2 = new TreeObject("Leaf 2");
        TreeObject to3 = new TreeObject("Leaf 3");
        TreeParent p1 = new TreeParent("Parent 1");
        p1.addChild(to1);
        p1.addChild(to2);
        p1.addChild(to3);

        TreeObject to4 = new TreeObject("Leaf 4");
        TreeParent p2 = new TreeParent("Parent 2");
        p2.addChild(to4);

        TreeParent root = new TreeParent("Root");
        root.addChild(p1);
        root.addChild(p2);

        invisibleRoot = new TreeParent("");
        invisibleRoot.addChild(root);
    }
}
//实现标签提供器
class ViewLabelProvider extends LabelProvider {

    public String getText(Object obj) {
        return obj.toString();
    }
    public Image getImage(Object obj) {
        String imageKey = ISharedImages.IMG_OBJ_ELEMENT;
        if (obj instanceof TreeParent)
          imageKey = ISharedImages.IMG_OBJ_FOLDER;
        return PlatformUI.getWorkbench().getSharedImages().getImage(imageKey);
    }
}
class NameSorter extends ViewerSorter {
}

public TreeViewTest() {
}
```

```
public void createPartControl(Composite parent) {
    viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
    drillDownAdapter = new DrillDownAdapter(viewer);
    viewer.setContentProvider(new ViewContentProvider());
    viewer.setLabelProvider(new ViewLabelProvider());
    viewer.setSorter(new NameSorter());
    viewer.setInput(getViewSite());
    makeActions();
    hookContextMenu();
    hookDoubleClickAction();
    contributeToActionBars();
}
//提供右键菜单
private void hookContextMenu() {
    MenuManager menuMgr = new MenuManager("#PopupMenu");
    menuMgr.setRemoveAllWhenShown(true);
    menuMgr.addMenuListener(new IMenuListener() {
        public void menuAboutToShow(IMenuManager manager) {
            TreeViewTest.this.fillContextMenu(manager);
        }
    });
    Menu menu = menuMgr.createContextMenu(viewer.getControl());
    viewer.getControl().setMenu(menu);
    getSite().registerContextMenu(menuMgr, viewer);
}
//提供视图工具栏
private void contributeToActionBars() {
    IActionBars bars = getViewSite().getActionBars();
    fillLocalPullDown(bars.getMenuManager());
    fillLocalToolBar(bars.getToolBarManager());
}

private void fillLocalPullDown(IMenuManager manager) {
    manager.add(action1);
    manager.add(new Separator());
    manager.add(action2);
}
//填充菜单
private void fillContextMenu(IMenuManager manager) {
    manager.add(action1);
    manager.add(action2);
    manager.add(new Separator());
    drillDownAdapter.addNavigationActions(manager);
    // Other plug-ins can contribute there actions here
    manager.add(new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
}

private void fillLocalToolBar(IToolBarManager manager) {
    manager.add(action1);
    manager.add(action2);
```

```
        manager.add(new Separator());
        drillDownAdapter.addNavigationActions(manager);
    }
    //创建菜单和工具栏对应的Action
    private void makeActions() {
        action1 = new Action() {
            public void run() {
                showMessage("Action 1 executed");
            }
        };
        action1.setText("Action 1");
        action1.setToolTipText("Action 1 tooltip");
        action1.setImageDescriptor(PlatformUI.getWorkbench().getSharedImages().
            getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));

        action2 = new Action() {
            public void run() {
                showMessage("Action 2 executed");
            }
        };
        action2.setText("Action 2");
        action2.setToolTipText("Action 2 tooltip");
        action2.setImageDescriptor(PlatformUI.getWorkbench().getSharedImages().
                getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));
        doubleClickAction = new Action() {
            public void run() {
                ISelection selection = viewer.getSelection();
                Object obj = ((IStructuredSelection)selection).getFirstElement();
                showMessage("Double-click detected on "+obj.toString());
            }
        };
    }

    private void hookDoubleClickAction() {
        viewer.addDoubleClickListener(new IDoubleClickListener() {
            public void doubleClick(DoubleClickEvent event) {
                doubleClickAction.run();
            }
        });
    }
    private void showMessage(String message) {
        MessageDialog.openInformation(
            viewer.getControl().getShell(),
            "Tree View",
            message);
    }
    //响应获得焦点
    public void setFocus() {
        viewer.getControl().setFocus();
    }
}
```

上例中通过内容提供器（ViewContentProvider）、标签提供器（ViewLabelProvider）及树节点（TreeObject）的定义提供了树的实现，另外通过 createPartControl 方法实现了相应的菜单和工具栏。

本例中右键菜单和工具栏的实现方式非常值得效仿，而且非常直观，读者可以比较一下在 SWT 中的实现方式，找到它们之间的异同点。

13.3.4 运行“Tree viewer”视图

“Tree viewer”插件到此就已经完成，现在可以调试插件了。

（1）运行插件。

（2）打开运行时工作台的“Show View”。

在“Show View”窗口中可以看到前面建立的分类和视图节点，如图 13-12 所示。

（3）运行“Tree viewer”视图

双击“Tree View”节点，打开视图窗口，如图 13-13 所示。



图 13-12 “Show View”窗口



图 13-13 “Tree viewer”视图

13.4 “Image viewer”视图

上一节的视图通过动态的方式建立菜单和工具栏选项，在 Eclipse 中也可以通过静态的方式生成。本节的目的就是要通过一个图片浏览的例子，介绍如何在 Eclipse 中扩展“org.eclipse.ui.viewActions”扩展点建立工具栏选项。

13.4.1 创建“Image viewer”视图插件

创建“Image viewer”视图插件的步骤如下。

（1）创建一个空插件项目，参考 13.2 节。

（2）添加“Image viewer”要实现的扩展点，如图 13-14 所示。

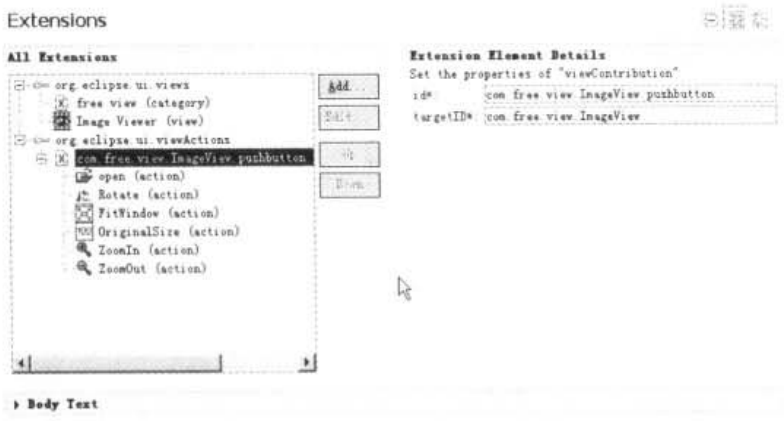


图 13-14　“Image viewer”插件扩展点

“Image viewer”插件实现了两个扩展点“org.eclipse.ui.views”和“org.eclipse.ui.viewActions”。插件 plugin.xml 文件如例程 13-5 所示。

例程 13-5　plugin.xml

```
<plugin>
<extension
      point="org.eclipse.ui.views">
    <category
        name="free view"
        id="com.free.view.categoryall">
    </category>
    <view
        name="Image Viewer"
        icon="icons/view.ico"
        category="com.free.view.categoryall"
        class="com.free.view.imageview.views.ImageView"
        id="com.free.view.ImageView">
    </view>
  </extension>

<!-- 工具栏按钮 -->
  <extension
      point="org.eclipse.ui.viewActions">
    <viewContribution
        id="com.free.view.ImageView.pushbutton"
        targetID="com.free.view.ImageView">
      <!--  打开按钮 -->
      <action
          label="open"
          icon="icons/Open16.gif"
          tooltip="Open image"
          class="com.free.view.imageview.actions.PushActionDelegate"
          toolbarPath="push_group"
          enablesFor="*"
          id="toolbar.open">
          <!--id="org.eclipse.ui.file.open"-->
```

```
</action>
<!-- 旋转按钮 -->
<action
     label="Rotate"
     icon="icons/Rotate16.gif"
     tooltip="Rotate"
     class="com.free.view.imageview.actions.PushActionDelegate"
     toolbarPath="push_group"
     enablesFor="*"
     id="toolbar.rotate">
</action>
<!-- 图片适配按钮 -->
<action
     label="FitWindow"
     icon="icons/Fit16.gif"
     tooltip="Fit window"
     class="com.free.view.imageview.actions.PushActionDelegate"
     toolbarPath="push_group"
     enablesFor="*"
     id="toolbar.fit">
</action>
<!-- 图片原始大小按钮 -->
<action
     label="OriginalSize"
     icon="icons/Original16.gif"
     tooltip="Original size"
     class="com.free.view.imageview.actions.PushActionDelegate"
     toolbarPath="push_group"
     enablesFor="*"
     id="toolbar.original">
</action>
<!-- 图片缩小按钮 -->
<action
     label="ZoomIn"
     icon="icons/ZoomIn16.gif"
     tooltip="Zoom in"
     class="com.free.view.imageview.actions.PushActionDelegate"
     toolbarPath="push_group"
     enablesFor="*"
     id="toolbar.zoomin">
</action>
<!-- 图片放大按钮 -->
<action
     label="ZoomOut"
     icon="icons/ZoomOut16.gif"
     tooltip="Zoom out"
     class="com.free.view.imageview.actions.PushActionDelegate"
     toolbarPath="push_group"
     enablesFor="*"
     id="toolbar.zoomout">
</action>
```

```
    </viewContribution>
  </extension>

</plugin>
```

上例中实现了 View 扩展点和 Action 扩展点，并且把 Action 关联到工具栏的相应按钮，也就是说单击按钮就能够执行所关联的 Action。

13.4.2　“org.eclipse.ui.viewActions”扩展点的实现类“PushActionDelegate”

要通过扩展点的方式实现视图的 Action，用户要实现“org.eclipse.ui.viewActions”扩展点，Eclipse 将会通过 Action 指定的实现类处理相应的请求。“Image viewer”视图通过 PushActionDelegate 响应 Action。

PushActionDelegate 类要实现 IViewActionDelegate 接口才能够实现相应的 Action 请求。PushActionDelegate 类通过 run 方法得到相应的 ID 号执行相应操作，如例程 13-6 所示。

例程 13-6　PushActionDelegate.java

```
package com.free.view.imageview.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.swt.graphics.ImageData;
import org.eclipse.swt.graphics.PaletteData;
import org.eclipse.ui.IViewActionDelegate;
import org.eclipse.ui.IViewPart;

import com.free.view.imageview.views.ImageView;
import com.free.view.imageview.views.SWTImageCanvas;

public class PushActionDelegate implements IViewActionDelegate {
    public ImageView view=null;
    public String id;

    public void init(IViewPart viewPart) {
        if(viewPart instanceof ImageView){
            this.view=(ImageView)viewPart;
        }
    }

    public void run(IAction action) {
        String id=action.getId();
        SWTImageCanvas imageCanvas=view.imageCanvas;
        if(id.equals("toolbar.open")){
            imageCanvas.onFileOpen();
            return;
        }
        if(imageCanvas.getSourceImage()==null) return;
        if(id.equals("toolbar.zoomin")){
            imageCanvas.zoomIn();
```

```
            return;
        }else if(id.equals("toolbar.zoomout")){
            imageCanvas.zoomOut();
            return;
        }else if(id.equals("toolbar.fit")){
            imageCanvas.fitCanvas();
            return;
        }else if(id.equals("toolbar.rotate")){
            ImageData src=imageCanvas.getImageData();
            if(src==null) return;
            PaletteData srcPal=src.palette;
            PaletteData destPal;
            ImageData dest;
            if(srcPal.isDirect){
                destPal=new PaletteData(srcPal.redMask,srcPal.greenMask,
                srcPal.blueMask);
            }else{
                destPal=new PaletteData(srcPal.getRGBs());
            }
            dest=new ImageData(src.height,src.width,src.depth,destPal);
            for(int i=0;i<src.width;i++){
                for(int j=0;j<src.height;j++){
                    int pixel=src.getPixel(i,j);
                    dest.setPixel(j,src.width-1-i,pixel);
                }
            }
            imageCanvas.setImageData(dest);
            return;
        }else if(id.equals("toolbar.original")){
            imageCanvas.showOriginal();
            return;
        }
    }

    public void selectionChanged(IAction action, ISelection selection) {}

}
```

13.4.3 “org.eclipse.ui.views”扩展点的实现类“ImageView”

ImageView 类实现建立一个图片处理的 SWT 组件“SWTImageCanvas”，所有对图片的操作交由 SWTImageCanvas 类进行处理，如例程 13-7 所示。

例程 13-7 ImageView.java

```
package com.free.view.imageview.views;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

public class ImageView extends ViewPart {
```

```
    public SWTImageCanvas imageCanvas;

    public ImageView() {
    }

    public void createPartControl(Composite frame) {
        imageCanvas=new SWTImageCanvas(frame);
    }

    public void setFocus() {
        imageCanvas.setFocus();
    }

    public void dispose() {
        imageCanvas.dispose();
        super.dispose();
    }

}
```

上例 ImageView 实现了 ViewPart，在 createPartControl 中通过 SWTImageCanvas 类处理图片的显示。

13.4.4　运行“Image viewer”视图

“Image viewer”视图的图片处理类“SWTImageCanvas”在这里就不做详细介绍，用户可以在本书附带的光盘中找到相应的代码。图 13-15 所示为“Iimage viewer”运行效果图。



图 13-15　“Image viewer”运行效果图

13.5 Debug 视图的实现

通过前面几节的实践，读者应该能够通过向导或通过模板建立自己的视图。本节的主要目的是让读者能够查找 Eclipse 中比较感兴趣的视图，并实现类似的视图。

13.5.1 查找 Debug 插件

Eclipse 中已经实现了很多视图，其中 Debug 方式下就有很多个视图的实现，如图 13-16 所示。

为了查找 Eclipse 中哪些插件实现视图，可以打开查找对话框，查找有哪些插件实现了“org.eclipse.ui.views”扩展点，如图 13-17 所示。



图 13-16 选择 Debug 视图

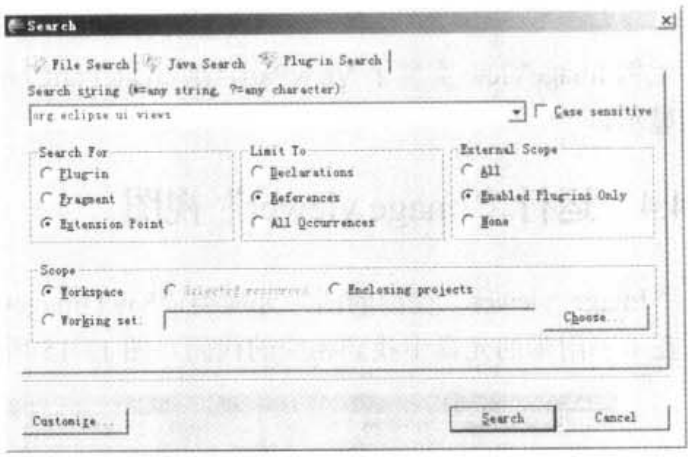


图 13-17 查找对话框

选择“Plug-in Search”选项卡，表示在插件中查找。在查找结果中，可以看到“org.eclipse.debug.ui”插件实现了“org.eclipse.ui.views”扩展点。

13.5.2 导入 Debug 插件

为了研究“org.eclipse.debug.ui”插件，首先需要把“org.eclipse.debug.ui”插件作为项目导入到 Workspace 中。导入插件的步骤如下。

（1）选择 Plug-in 视角。

（2）选择 Plug-ins 浏览视图，如图 13-18 所示。

（3）导入“org.eclipse.debug.ui”插件。

Eclipse 所在注册的插件都会显示在 Plug-ins 浏览视图中，在 Plug-ins 浏览视图中选择“org.eclipse.debug.ui”插件。单击鼠标右键选择导入源文件项目，在此选择以源代码的方式导入，如图 13-19 所示。

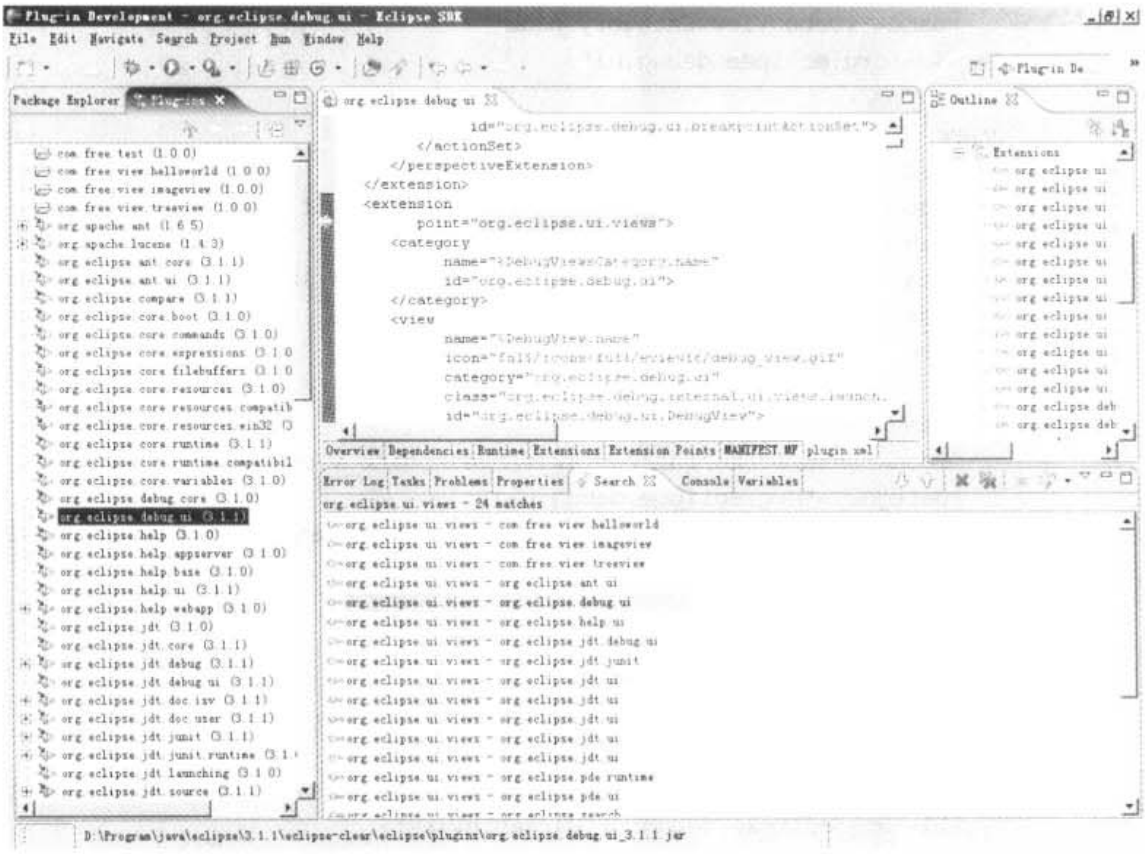


图 13-18　Plug-ins 浏览视图



图 13-19　导入插件

13.5.3　Debug 插件中视图的扩展点

打开“org.eclipse.debug.ui”的 plugin.xml 文件，可以看到 Debug 插件实现了几个视图，如例程 13-8 所示。

例程 13-8　plugin.xml

```
...
<extension
      point="org.eclipse.ui.views">
   <category
```

```
        name="%DebugViewsCategory.name"
        id="org.eclipse.debug.ui">
    </category>
    <view
        name="%DebugView.name"
        icon="$nl$/icons/full/eview16/debug_view.gif"
        category="org.eclipse.debug.ui"
        class="org.eclipse.debug.internal.ui.views.launch.LaunchView"
        id="org.eclipse.debug.ui.DebugView">
    </view>
    <view
        name="%BreakpointsView.name"
        icon="$nl$/icons/full/eview16/breakpoint_view.gif"
        category="org.eclipse.debug.ui"
        class="org.eclipse.debug.internal.ui.views.breakpoints.BreakpointsView"
        id="org.eclipse.debug.ui.BreakpointView">
    </view>
    <view
        name="%VariablesView.name"
        icon="$nl$/icons/full/eview16/variable_view.gif"
        category="org.eclipse.debug.ui"
        class="org.eclipse.debug.internal.ui.views.variables.VariablesView"
        id="org.eclipse.debug.ui.VariableView">
    </view>
    <view
        name="%ExpressionView.name"
        icon="$nl$/icons/full/eview16/watchlist_view.gif"
        category="org.eclipse.debug.ui"
        class="org.eclipse.debug.internal.ui.views.expression.ExpressionView"
        id="org.eclipse.debug.ui.ExpressionView">
    </view>
    <view
        name="%RegistersView.name"
        icon="$nl$/icons/full/eview16/register_view.gif"
        category="org.eclipse.debug.ui"
        class="org.eclipse.debug.internal.ui.views.registers.RegistersView"
        id="org.eclipse.debug.ui.RegisterView">
    </view>
    <view
        allowMultiple="true"
        category="org.eclipse.debug.ui"
        class="org.eclipse.debug.internal.ui.views.memory.MemoryView"
        icon="$nl$/icons/full/eview16/memory_view.gif"
        id="org.eclipse.debug.ui.MemoryView"
        name="%MemoryViewName">
    </view>
</extension>
```

Eclipse 很多开源的插件都会把源代码放在插件中（插件打包时可选择是否把源代码放入插件包中），可以通过这种方式导入插件项目。

另外，读者如果对 Debug 插件的视图实现感兴趣，可以研究导入的项目源代码，这也是研究 Eclipse 中优秀代码的很好方式。

13.6　小　　结

本章介绍了视图的概念，以及通过手动和模板的方式加入视图的扩展点，最后介绍了怎样导入 Debug 插件。

读者应该掌握如下几个概念。

- 如何在 plugin.xml 声明视图的扩展点。
- 如何在实现类的 createPartControl 方法中构造视图的内容。
- 如何在视图中加入菜单和工具栏。
- 如何查找并导入 Eclipse 的视图。

第 14 章　编辑器扩展点的实现

编辑器和视图都是 Eclipse 中可视的组件，Eclipse 中编辑的主要功能是编辑资源，用户可以通过实现编辑器扩展点实现编辑器，编辑自己感兴趣的资源。

本章主要从以下几个部分进行介绍。

- Editors 扩展点的介绍
- “Hello，world”编辑器
- 多页签编辑器的实现
- 简单的文件编辑器

14.1　Editors 扩展点的介绍

编辑器是工作台页面内的可视组件，它通常用来编辑或浏览文档或输入对象。要打开编辑器，用户通常将对 IFile 调用“打开”。当执行此操作时，将查阅工作台注册表以便为文件类型确定适当的编辑器，然后创建编辑器类型的新实例，实际结果取决于编辑器的类型。

工作台能够创建内部编辑器（它已紧密集成到工作台中）和外部编辑器（它是在单独的框架窗口中启动的）。这些编辑器之间还具有各种级别的集成。用户可以扩展“org.eclipse.ui.editors”扩展点实现自己的编辑器。

14.1.1　编辑器分类

Eclipse 中同一时间只有一个编辑器可见（视图可以有多个），但能打开同一编辑器的多个实例（视图只有一个实例）。编辑器一般来说有两种类型。

- 文本编辑器：文本编辑器负责对文本的编辑，通常所见的 Java 文件编辑器就是文本编辑器。
- 图形编辑器：图形编辑器负责对图形的编辑，例如 GEF 实现的图形编辑器（后面章节会介绍 GEF 实现的图形编辑器）。

14.1.2　常用编辑器可实现的功能

常用编辑器具有的功能如下。

- 打开文件
- 保存文件
- 编辑文件
- 撤销、重做

另外，编辑器还能够根据文件的内容初始化编辑区域的显示，例如插件的清单文件是

个文本文件，编辑器会根据文件的内容决定显示的方式。

14.1.3 编辑器和视图的区别

编辑器和视图都是工作台页面中显示的可视化组件，两者间区别如下。

- 保存：在视图中所做的修改将被立即反映到工作区，而编辑器则有一个“打开—修改—保存”的生命周期。也就是说，只有当用户明确执行保存操作时，在编辑器中所做的修改才会被保存。
- 输入：编辑器是由用户选中特定元素后打开的，这个元素就成为了编辑器的输入（每个输入不同，所以编辑器编辑的内容就不同）。视图没有预定的输入，每个视图各自决定到哪里去获得自己的输入数据。
- 用户界面：视图有自己的本地工具栏，编辑器则只有全局的工具栏。编辑器只能在工作台的“主编辑器”区域出现，视图则可以在工作台中任意摆放，唯一的限制是不能放在编辑器的前面或者后面。
- 实例数量：在一个工作台页面中，一个特定类型的视图最多只会出现一次。而同一类型的编辑器可以在同一个工作台页面中打开多次，各个编辑器实例对应不同的输入。

14.2 “Hello，world”编辑器

本节通过“Hello，world”编辑器让读者了解建立一个编辑器的步骤，“Hello，world”编辑器只在编辑器显示“hello, world”标签。

14.2.1 “Hello，world”编辑器插件

通过 Eclipse 的向导建立一个插件项目，步骤如下。

（1）打开新建向导。

单击 Eclipse 的“File”→“New”→“Other”菜单项，在弹出的对话框中选择 Select 对话框左边的 Plug-in Development 向导。如图 14-1 所示，选择 Plug-in Project。

（2）建立一个空的插件项目。

① 输入插件工程的名字为“com.free.editor.helloworld”。

② 输入插件的 ID 为“com.free. editor.helloworld”。

③ 输入插件的 Name 为“Helloworld Editor Plug-in”。

④ 不选择使用模板建立插件项目，如图 14-2 所示。

⑤ 单击“Finish”按钮，完成项目的建立。

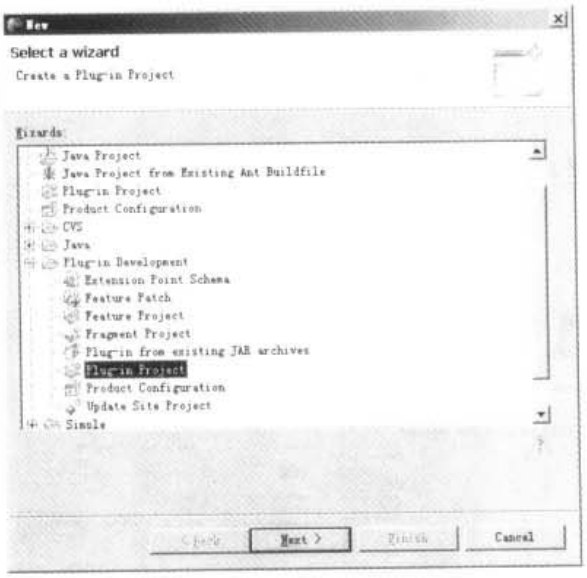


图 14-1　新建插件对话框

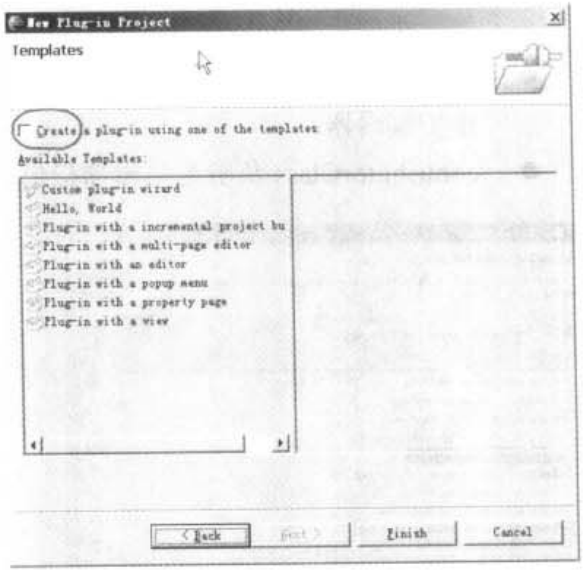


图 14-2　选择插件模板对话框

通过这种方式建立了一个空的插件项目，要让此插件成为一个视图的插件，还要实现相应的扩展点。

14.2.2　“Hello，world”插件扩展点

为了实现视图插件，就应该实现 Eclipse 提供的视图扩展点，具体步骤如下。

（1）打开插件描述文件的 Extensions 页。

新建一个插件后，会自动打开插件清单文件编辑器，读者也可以通过 META-INF 目录下的 MENIFEST.MF 文件打开插件清单文件编辑器。

（2）单击“Add”按钮。

（3）选择“org.eclipse.ui.editors”扩展点。

如图 14-3 所示，选择“org.eclipse.ui. editors”扩展点，单击“Finish”按钮。读者也可以通过向导选择模板生成相应的扩展点清单文件，在此笔者采用手动方式生成插件清单文件。

（4）新建一个编辑器节点（editor）。

新建一个编辑器节点，并对所选节点赋值，如图 14-4 所示。

- id：“com.free.editor.helloworld.editor”。
- name：“hello world! editor”。
- icon：“icons/sample.gif”。
- extensions：“hello”。
- class:“com.free.editor.editor. HelloWorldEditor”。
- contributorClass：“空”。

编辑器节点值的解释。

- id 为编辑器的编号，用户可以通过此编号在程序中访问该视图。
- name 为标识编辑器的名字。
- icon 为编辑器的显示图标。
- extensions 为编辑器对应的文件扩展名（Eclipse 中会通过扩展名打开相应注册的编辑

器，类似于 Windows 下的文件关联）。

- class 为编辑的实现类，当用户选择打开编辑器时，Eclipse 会自动调用此类初始化并维护编辑器。
- contributorClass 负责为编辑器提供工具栏和菜单（本例中未实现 contributorClass）。

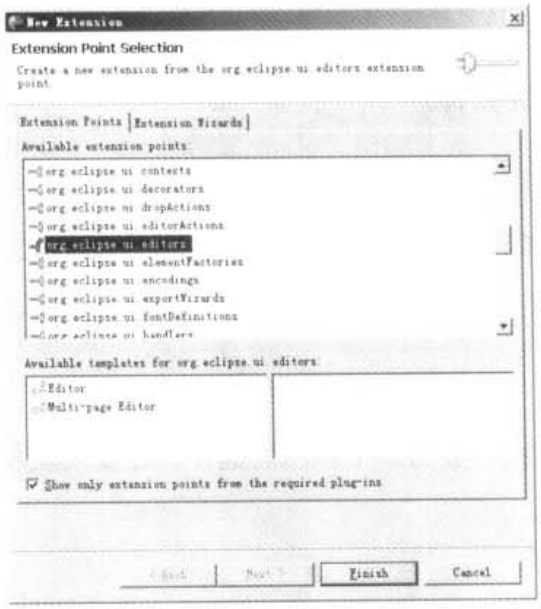


图 14-3　选择扩展点

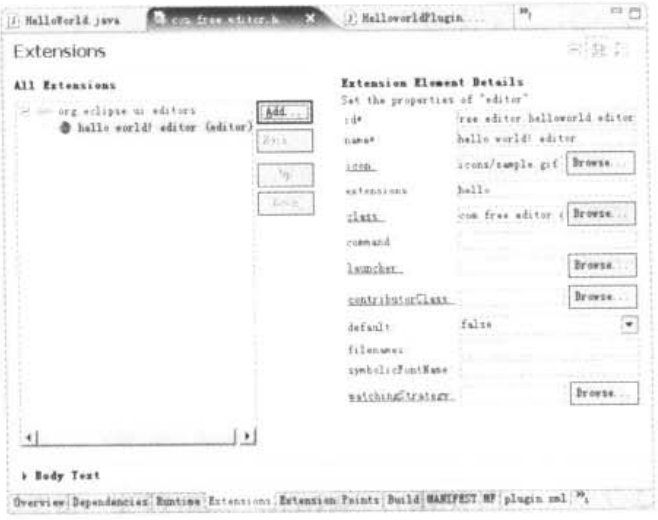


图 14-4　编辑器节点

“Hello，world”编辑器的 plugin.xml 文件，如例程 14-1 所示。

例程 14-1　plugin.xml

```
<plugin>
  <extension
      point="org.eclipse.ui.editors">
    <editor
        class="com.free.editor.editor.HelloWorldEditor"
        default="false"
        extensions="hello"
        icon="icons/sample.gif"
        id="com.free.editor.helloworld.editor"
        name="hello world! editor"/>
  </extension>
</plugin>
```

14.2.3　“org.eclipse.ui.editors”扩展点的实现类“HelloWorldEditor”

当用户选择视图后，Eclipse 会自动调用编辑器节点 Class 所指向的类对编辑器进行初始化。

用户定义的视图继承于“org.eclipse.ui.part.EditorPart”，且必须实现 createPartControl (Composite parent)方法、setFocus()方法、doSave(IProgressMonitor monitor)方法、doSaveAs()方法、init(IEditorSite site, IEditorInput input)方法、isDirty()方法和 isSaveAsAllowed()方法，如例程 14-2 所示。

例程 14-2　HelloWorldEditor.java

```
package com.free.editor.editor;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IEditorSite;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.part.EditorPart;

public class HelloWorldEditor extends EditorPart {

    public HelloWorldEditor() {
        super();
    }

    public void doSave(IProgressMonitor monitor) {

    }

    public void doSaveAs() {

    }

    public void init(IEditorSite site, IEditorInput input) throws PartInitException {
        setSite(site);
        setInput(input);

        setPartName("hello, world");
    }

    public boolean isDirty() {
        return false;
    }

    public boolean isSaveAsAllowed() {
        return false;
    }

    public void createPartControl(Composite parent) {
      Label label = new Label(parent, SWT.LEFT);
      label.setText("Hello, world!");

    }

    public void setFocus() {

    }
```

```
}
```

EditorPart 是一个抽象类，要实现 Eclipse 的编辑器，就必须实现 EditorPart 类的抽象方法，编辑器实现类中要实现 EditorPart 中的如下方法。

- doSave(IProgressMonitor monitor)：单击“保存”按钮将调用此方法。
- doSaveAs()：单击“另存为”菜单将调用此方法。
- init(IEditorSite site, IEditorInput input)：打开编辑器将调用此方法初始化编辑器。
- isDirty()：Eclipse 调用此方法判断编辑器中的数据是否被修改过。
- isSaveAsAllowed()：Eclipse 调用此方法判断编辑器中的数据是否可以另存为。
- createPartControl(Composite parent)：Eclipse 通过此方法构建编辑器的组件。
- setFocus()：编辑器激活时将调用此方法。

14.2.4 运行“Hello，world”编辑器

“Hello，world”插件到此已经完成，现在可以调试插件了。

（1）运行插件。

（2）新建一个项目，并在项目中新建一个扩展名为 hello 的文件。

在新建的项目中新建一个文件“test.hello”，hello 为“Hello，world”编辑器关联的扩展名，如图 14-5 所示。

（3）单击“Finish”按钮，打开“Hello，world”编辑器。

新建扩展名为 hello 的文件后将打开“Hello，world”编辑器，如图 14-6 所示。

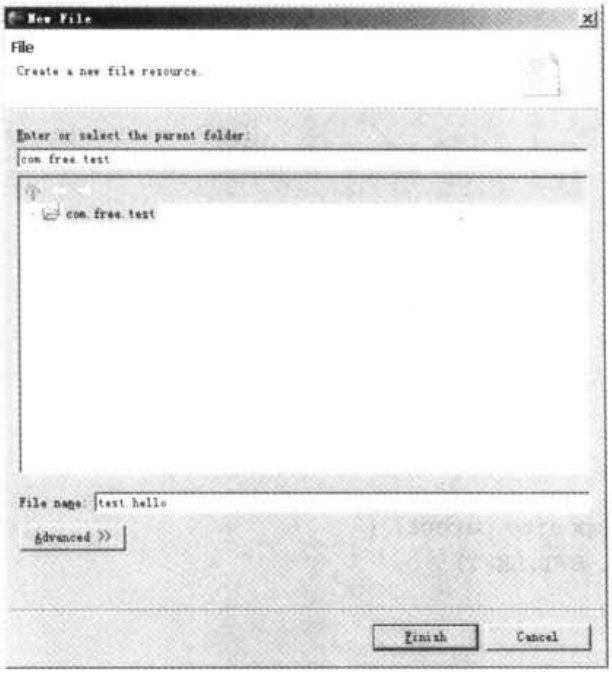


图 14-5 新建文件对话框

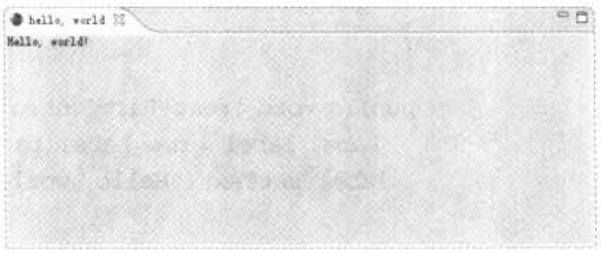


图 14-6 “Hello，world”编辑器

14.3　多页签编辑器的实现

上一节的“Hello，world”编辑器只是实现一个编辑器的扩展点，并未实现编辑器的可编辑功能，本节将通过 Eclipse 的插件模板向导建立一个有多页签编辑器的插件。

14.3.1　创建多页签编辑器插件

创建多页签编辑器插件，步骤如下。

（1）打开新建向导。

单击 Eclipse 的“File”→“New”→“Other”菜单项，在弹出的对话框中选择 Select 对话框左边的 Plug-in Development 向导。

（2）建立一个空的插件项目。

① 输入插件工程的名字为“com.free.editor. multipageeditor”。

② 输入插件的 ID 为“com.free.editor.multipageeditor”。

③ 输入插件的 Name 为“Multipageeditor Plug-in”。

选择使用模板建立插件项目，如图 14-7 所示。

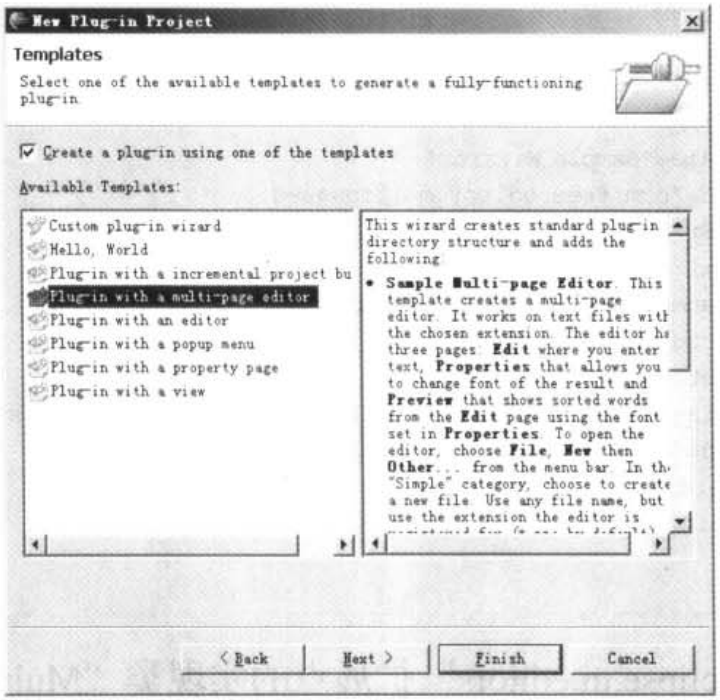


图 14-7　选择插件模板对话框

单击“Finish”按钮，完成项目的建立。

通过这种方式建立了一个包含多页签编辑器的插件项目，此项目已经具备了一般编辑器的常用功能。

14.3.2 多页签编辑器插件扩展点

多页签编辑器插件实现了“org.eclipse.ui.editors”和“org.eclipse.ui.newWizards”两个扩展点，本节只介绍 editors 扩展点，newWizards 扩展点在后面的章节会介绍到。插件 Plugin.xml 文件如例程 14-3 所示。

例程 14-3 plugin.xml

```
<plugin>

  <extension
      point="org.eclipse.ui.editors">
    <editor
        name="Sample Multi-page Editor"
        extensions="mpe"
        icon="icons/sample.gif"
        contributorClass=
            "com.free.editor.multipageeditor.editors.MultiPageEditorContributor"
        class="com.free.editor.multipageeditor.editors.MultiPageEditor"
        id="com.free.editor.multipageeditor.editors.MultiPageEditor">
    </editor>
  </extension>
  <extension
      point="org.eclipse.ui.newWizards">
    <category
        name="Sample Wizards"
        id="com.free.editor.multipageeditor">
    </category>
    <wizard
        name="Multi-page Editor file"
        icon="icons/sample.gif"
        category="com.free.editor.multipageeditor"
        class="com.free.editor.multipageeditor.wizards.SampleNewWizard"
        id="com.free.editor.multipageeditor.wizards.SampleNewWizard">
    </wizard>
  </extension>

</plugin>
```

14.3.3 “org.eclipse.ui.editors”扩展点的实现类“MultiPageEditor”

实现类中要完成如下的功能。

（1）初始化编辑器。

（2）建立多页编辑器。

（3）响应多页签之间的切换。

（4）对资源进行监听。

多页签编辑器继承自“MultiPageEditorPart”类，通过重载 createPages()方法实现多页签，

代码如下。

```
protected void createPages() {
        createPage0();
        createPage1();
        createPage2();
    }
```

多页签的主要功能是实现分页，在每个分页中，用户还要实现自己的编辑器，多页签编辑器中实现类，如例程 14-4 所示。

例程 14-4　MultiPageEditor.java

```
package com.free.editor.multipageeditor.editors;

import java.io.StringWriter;
import java.text.Collator;
import java.util.ArrayList;
import java.util.Collections;
import java.util.StringTokenizer;

import org.eclipse.core.resources.IMarker;
import org.eclipse.core.resources.IResourceChangeEvent;
import org.eclipse.core.resources.IResourceChangeListener;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.dialogs.ErrorDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.custom.StyledText;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.graphics.FontData;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.FontDialog;
import org.eclipse.ui.*;
import org.eclipse.ui.editors.text.TextEditor;
import org.eclipse.ui.part.FileEditorInput;
import org.eclipse.ui.part.MultiPageEditorPart;
import org.eclipse.ui.ide.IDE;

public class MultiPageEditor extends MultiPageEditorPart implements
IResourceChangeListener{

    private TextEditor editor;

    private Font font;
```

```
private StyledText text;

public MultiPageEditor() {
    super();
    ResourcesPlugin.getWorkspace().addResourceChangeListener(this);
}

void createPage0() {
    try {
        editor = new TextEditor();
        int index = addPage(editor, getEditorInput());
        setPageText(index, editor.getTitle());
    } catch (PartInitException e) {
        ErrorDialog.openError(
            getSite().getShell(),
            "Error creating nested text editor",
            null,
            e.getStatus());
    }
}

void createPage1() {

    Composite composite = new Composite(getContainer(), SWT.NONE);
    GridLayout layout = new GridLayout();
    composite.setLayout(layout);
    layout.numColumns = 2;

    Button fontButton = new Button(composite, SWT.NONE);
    GridData gd = new GridData(GridData.BEGINNING);
    gd.horizontalSpan = 2;
    fontButton.setLayoutData(gd);
    fontButton.setText("Change Font...");

    fontButton.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent event) {
            setFont();
        }
    });

    int index = addPage(composite);
    setPageText(index, "Properties");
}

void createPage2() {
    Composite composite = new Composite(getContainer(), SWT.NONE);
    FillLayout layout = new FillLayout();
    composite.setLayout(layout);
    text = new StyledText(composite, SWT.H_SCROLL | SWT.V_SCROLL);
    text.setEditable(false);
```

```
        int index = addPage(composite);
        setPageText(index, "Preview");
    }
    //建立多页签
    protected void createPages() {
        createPage0();
        createPage1();
        createPage2();
    }

    public void dispose() {
        ResourcesPlugin.getWorkspace().removeResourceChangeListener(this);
        super.dispose();
    }

    public void doSave(IProgressMonitor monitor) {
        getEditor(0).doSave(monitor);
    }
    //另存为
    public void doSaveAs() {
        IEditorPart editor = getEditor(0);
        editor.doSaveAs();
        setPageText(0, editor.getTitle());
        setInput(editor.getEditorInput());
    }

    public void gotoMarker(IMarker marker) {
        setActivePage(0);
        IDE.gotoMarker(getEditor(0), marker);
    }
    //初始化
    public void init(IEditorSite site, IEditorInput editorInput)
        throws PartInitException {
        if (!(editorInput instanceof IFileEditorInput))
            throw new PartInitException("Invalid Input: Must be IFileEditorInput");
        super.init(site, editorInput);
    }
    //编辑器是否允许另存为操作
    public boolean isSaveAsAllowed() {
        return true;
    }
    //相应多页签切换
    protected void pageChange(int newPageIndex) {
        super.pageChange(newPageIndex);
        if (newPageIndex == 2) {
            sortWords();
        }
    }
    //监听资源改变
    public void resourceChanged(final IResourceChangeEvent event){
```

```
        if(event.getType() == IResourceChangeEvent.PRE_CLOSE){
            Display.getDefault().asyncExec(new Runnable(){
                public void run(){
                    IWorkbenchPage[] pages = getSite().getWorkbenchWindow().
                    getPages();
                    for (int i = 0; i<pages.length; i++){
                        if(((FileEditorInput)editor.getEditorInput()).
                        getFile().getProject().equals(event.getResource())){
                            IEditorPart editorPart = pages[i].findEditor
                            (editor.getEditorInput());
                            pages[i].closeEditor(editorPart,true);
                        }
                    }
                }
            });
        }
    }

    void setFont() {
        FontDialog fontDialog = new FontDialog(getSite().getShell());
        fontDialog.setFontList(text.getFont().getFontData());
        FontData fontData = fontDialog.open();
        if (fontData != null) {
            if (font != null)
                font.dispose();
            font = new Font(text.getDisplay(), fontData);
            text.setFont(font);
        }
    }

    void sortWords() {

        String editorText =
            editor.getDocumentProvider().getDocument(editor.getEditorInput()).get();

        StringTokenizer tokenizer =
            new StringTokenizer(editorText, " \t\n\r\f!@#\u0024%^&*()-_=+`
            ~[]{};:'\",.<>/?|\\");
        ArrayList editorWords = new ArrayList();
        while (tokenizer.hasMoreTokens()) {
            editorWords.add(tokenizer.nextToken());
        }

        Collections.sort(editorWords, Collator.getInstance());
        StringWriter displayText = new StringWriter();
        for (int i = 0; i < editorWords.size(); i++) {
            displayText.write(((String) editorWords.get(i)));
            displayText.write(System.getProperty("line.separator"));
        }
        text.setText(displayText.toString());
    }
```

```
}
```

14.3.4　“org.eclipse.ui.editors”扩展点的辅助类“MultiPageEditorContributor”

contributeClass 负责为编辑器定制工具栏和菜单。Eclipse 中同一编辑器的不同实例共享工具栏和菜单的 Action。contributeClass 继承于 MultiPageEditorActionBarContributor 类。MultiPageEditorContributor 代码如例程 14-5 所示。

contributeClass 中方法解释如下。

- setActivePage(IEditorPart part)：切换编辑器调用此过程。
- contributeToMenu(IMenuManager manager)：建立编辑器对应菜单项。
- contributeToToolBar(IToolBarManager manager)：建立编辑器对应工具栏。

例程 14-5　MultiPageEditorContributor.java

```
package com.free.editor.multipageeditor.editors;

import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.IActionBars;
import org.eclipse.ui.IEditorPart;
import org.eclipse.ui.IWorkbenchActionConstants;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.actions.ActionFactory;
import org.eclipse.ui.ide.IDE;
import org.eclipse.ui.ide.IDEActionFactory;
import org.eclipse.ui.part.MultiPageEditorActionBarContributor;
import org.eclipse.ui.texteditor.ITextEditor;
import org.eclipse.ui.texteditor.ITextEditorActionConstants;

public class MultiPageEditorContributor extends MultiPageEditorActionBarContributor {
    private IEditorPart activeEditorPart;
    private Action sampleAction;

    public MultiPageEditorContributor() {
        super();
        createActions();
    }

    protected IAction getAction(ITextEditor editor, String actionID) {
        return (editor == null ? null : editor.getAction(actionID));
    }
    //切换编辑器调用此过程
    public void setActivePage(IEditorPart part) {
        //如果当前的活动 Part 和上一次的 Part 相同（即未切换编辑器），则返回
        if (activeEditorPart == part)
            return;

        activeEditorPart = part;
```

```
        IActionBars actionBars = getActionBars();
        if (actionBars != null) {

            ITextEditor editor = (part instanceof ITextEditor) ? (ITextEditor)
             part : null;

            actionBars.setGlobalActionHandler(
                ActionFactory.DELETE.getId(),
                getAction(editor, ITextEditorActionConstants.DELETE));
            actionBars.setGlobalActionHandler(
                ActionFactory.UNDO.getId(),
                getAction(editor, ITextEditorActionConstants.UNDO));
            actionBars.setGlobalActionHandler(
                ActionFactory.REDO.getId(),
                getAction(editor, ITextEditorActionConstants.REDO));
            actionBars.setGlobalActionHandler(
                ActionFactory.CUT.getId(),
                getAction(editor, ITextEditorActionConstants.CUT));
            actionBars.setGlobalActionHandler(
                ActionFactory.COPY.getId(),
                getAction(editor, ITextEditorActionConstants.COPY));
            actionBars.setGlobalActionHandler(
                ActionFactory.PASTE.getId(),
                getAction(editor, ITextEditorActionConstants.PASTE));
            actionBars.setGlobalActionHandler(
                ActionFactory.SELECT_ALL.getId(),
                getAction(editor, ITextEditorActionConstants.SELECT_ALL));
            actionBars.setGlobalActionHandler(
                ActionFactory.FIND.getId(),
                getAction(editor, ITextEditorActionConstants.FIND));
            actionBars.setGlobalActionHandler(
                IDEActionFactory.BOOKMARK.getId(),
                getAction(editor, IDEActionFactory.BOOKMARK.getId()));
            actionBars.updateActionBars();
        }
    }
    private void createActions() {
        sampleAction = new Action() {
            public void run() {
                MessageDialog.openInformation(null, "Multipageeditor Plug-in",
                "Sample Action Executed");
            }
        };
        sampleAction.setText("Sample Action");
        sampleAction.setToolTipText("Sample Action tool tip");
        sampleAction.setImageDescriptor(PlatformUI.getWorkbench().
        getSharedImages().
                getImageDescriptor(IDE.SharedImages.IMG_OBJS_TASK_TSK));
    }
    //建立菜单
    public void contributeToMenu(IMenuManager manager) {
```

```
            IMenuManager menu = new MenuManager("Editor &Menu");
            manager.prependToGroup(IWorkbenchActionConstants.MB_ADDITIONS, menu);
            menu.add(sampleAction);
        }
        //建立工具栏
        public void contributeToToolBar(IToolBarManager manager) {
            manager.add(new Separator());
            manager.add(sampleAction);
        }
    }
```

14.3.5　运行多页签编辑器

多页签编辑器插件到此已经完成，现在可以调试插件了。

（1）运行插件。

（2）新建一个项目，并在项目中新建一个扩展名为 mpe 的文件。

（3）新建扩展名为 mpe 的文件后将打开多页签编辑器，如图 14-8 所示。



图 14-8　多页签编辑器

另外，用户还可以在打开编辑器实例的时候看到“MultiPageEditorContributor”实现的工具栏等。

14.4　简单的文本编辑器

前面已经介绍了如何构建一个多页签编辑器，在多页签编辑器中采用“TextEditor”作为文本编辑器，并且能对文本进行编辑，但如果用户要自己构建编辑器，那么就必须自己处理文件的输入和其他跟编辑相关的操作。本节将通过 SWT 构造一个简单的文本编辑器，引导读者完全手动构建一个编辑器，并且处理编辑器的输入和输出。

14.4.1　创建文本编辑器插件

创建文本编辑器插件步骤如下。

（1）创建一个空插件项目，参考 13.2 节。

（2）添加文本编辑器要实现的扩展点，如图 14-9 所示。

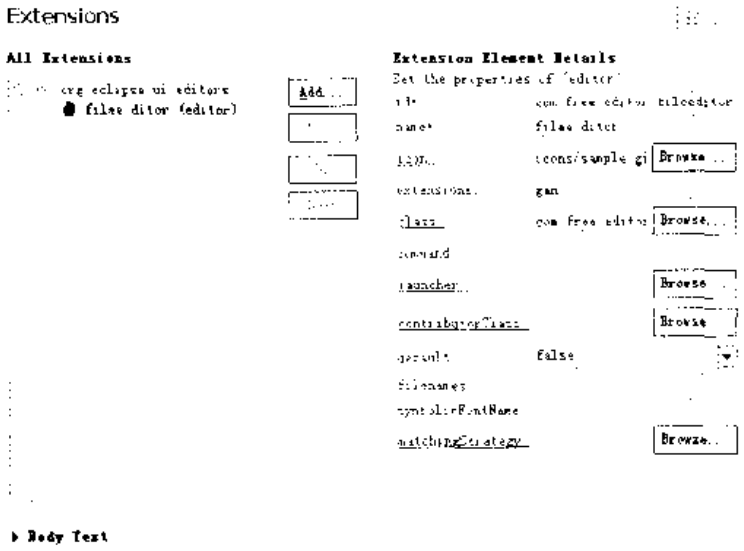


图 14-9　文件编辑器插件扩展点

文本编辑器插件实现了扩展点“org.eclipse.ui.editors”。插件 plugin.xml 文件如例程 14-6 所示。

例程 14-6　plugin.xml

```
<plugin>
  <extension
      point="org.eclipse.ui.editors">
    <editor
        class="com.free.editor.fileeditor.FileEditor"
        default="false"
        extensions="gan"
        icon="icons/sample.gif"
        id="com.free.editor.fileeditor"
        name="filee ditor"/>
  </extension>

</plugin>
```

14.4.2　“org.eclipse.ui.editors”扩展点的实现类“FileEditor”

FileEditor 负责初始化编辑器，根据文件的内容决定编辑器的显示方式。FileEditor 通过重载 setInput(IEditorInput input)方法获得输入信息，通过 doSave(IProgressMonitor monitor)方法保存编辑器中的内容，代码如例程 14-7 所示。

例程 14-7　FileEditor.java

```
package com.free.editor.fileeditor;

import java.io.ByteArrayInputStream;
import java.io.InputStream;
```

```
import org.eclipse.core.resources.IFile;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.swt.SWT;
import org.eclipse.swt.custom.StyledText;
import org.eclipse.swt.events.ModifyEvent;
import org.eclipse.swt.events.ModifyListener;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IEditorSite;
import org.eclipse.ui.IFileEditorInput;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.part.EditorPart;


public class FileEditor extends EditorPart {

    StyledText text = null;
    private String contentString = "";

    private boolean myDirty = false;

    public FileEditor() {
        super();
    }

    //保存编辑器内容
    public void doSave(IProgressMonitor monitor) {
        try {
            IFile file = ((IFileEditorInput) getEditorInput()).getFile();

            ByteArrayInputStream bi = new ByteArrayInputStream(text.getText()
                    .getBytes());
            file.setContents(bi, true, false, monitor);
            myDirty = false;
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    public void doSaveAs() {

    }

    //初始化编辑器
    public void init(IEditorSite site, IEditorInput input) throws PartInitException {
         setSite(site);
        setInput(input);

        setPartName("text editor");
```

```
    }
    //取得文件输入的信息
    public void setInput(IEditorInput input) {

        super.setInput(input);
        IFile file = ((IFileEditorInput) input).getFile();
        try {
            InputStream is = file.getContents(false);
            byte[] inputByte = new byte[is.available()];

            is.read(inputByte);

            contentString = new String(inputByte);

        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    //编辑器中内容是否被修改过
    public boolean isDirty() {
        return myDirty;
    }

    public boolean isSaveAsAllowed() {
        return false;
    }

    //构建编辑器的组件
    public void createPartControl(Composite parent) {

        text = new StyledText(parent, SWT.LEFT);
        text.setText(contentString);

        text.addModifyListener(new ModifyListener() {
            public void modifyText(ModifyEvent e) {
              myDirty = true;
            }
           });

    }

    public void setFocus() {

    }

}
```

FileEditor 实现的功能非常简单，但作为自定义编辑器的实例已经足够了。通过 FileEditor

能处理磁盘文件的输入，并显示信息，还能够保存编辑的内容到文件中。

14.4.3　运行文本编辑器

文本编辑器功能非常简单，主要目的是让读者了解如何获得编辑器的输入数据，读者也可以对数据进行解析构建自己的编辑器。文本编辑器运行效果如图 14-10 所示。



图 14-10　文本编辑器

14.5　小　　结

本章介绍了编辑器的基本概念，以及如何构建一个编辑器。通过对本章的学习读者应该了解如下几个方面的内容。

- Eclipse 中编辑器的概念。
- Eclipse 中编辑器和视图的区别。
- 通过扩展点构建编辑器。
- 构建编辑器的菜单和工具栏。
- 获得编辑器的输入数据。

第 15 章　项目相关扩展点的实现

Eclipse 中所有的开发都是基于插件的，用户如果想在 Eclipse 之上建立自己的平台，或者想把自己的项目集成到 Eclipse 中，使 Eclipse 更像是自己的产品，那么要涉及的扩展点就比较多了。本章以 Eclipse 项目为基础，介绍实现用户自定义项目要扩展的一些常用扩展点。

本章主要从以下几个部分进行介绍。

- 项目中常用的扩展
- 视角
- 菜单工具栏
- 项目相关特性

15.1　项目中常用的扩展

在 Eclipse 中，资源是以项目形式进行组织的，用户如果要集成 Eclipse 组织自己的资源，不可避免地要涉及到项目。

用户如果要扩展 Eclipse 插件，在 Eclipse 的基础上建立自己的产品，一般来说要实现如下几个部分的功能。

（1）用户自定义的视角
（2）用户自定义的视图
（3）用户自定义的编辑器
（4）加入项目特性
（5）项目构建器
（6）新建向导
（7）菜单和工具栏按钮
（8）首选页
（9）弹出菜单
（10）帮助

在前面章节，读者已经了解了视图、编辑器扩展点的实现，本章将会介绍如何创建用户的视角、菜单、工具栏按钮、项目特性、构建器和首选页，帮助和其他的一些扩展点将会在后面的章节介绍。

15.2　视　　角

Eclipse 中的视角是一个比较有趣的概念，也非常有用。想象一下，Eclipse 是一个插件的大拼图，在同一时间 Eclipse 如果实现了很多个视图，把这么多个视图同时展现给用户会是什

么场景。因为大多的视图同时展现出来，布局一定会很乱，结果肯定是用户不会用 Eclipse 了。Eclipse 中通过视角来组织视图的显示，解决了视图显示的问题。

15.2.1 简介

如果把所有的视图都在同一时间展现给用户肯定是不行的，一方面，Eclipse 在同一时间打开成千上万个视图的效率会很低（效率是 Eclipse 中的一个关注点），另一方面，如果在同一时间成千上万个视图堆在一块，肯定不美观（而且大多视图在某一时刻用户是不关心的）。

Eclipse 通过视角（有些书上也称为透视图）很好地解决了这个问题。就如同一个物体，这个物体有很多个面，从一个视角只能看到几个面。Eclipse 中同一个视角可以有若干个视图，用户可以通过视角扩展点建立自己的视角，在此视角中包含自己感兴趣的视图，那么当用户选择此视角时，Eclipse 就会自动打开此视角中包含的视图（从这个角度说，笔者觉得视角这个词较准确）。

可以通过“Window”→“Open perspectives”→“Other”打开选择视角的对话框，如图 15-1 所示。

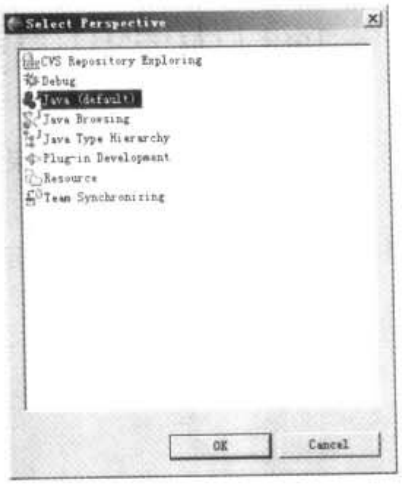


图 15-1 选择视角

在 Eclipse 中，用户能够扩展 Eclipse 的扩展点实现自己的视角，也可以实现扩展点把自己的视图加入到其他的视角中，后面将简单介绍如何通过这两个方法实现视角。

> **注意** 视角只包含视图的引用，当选择了视角后，此视角会自动打开视角中包含的视图引用。如果视图还未初始化，Eclipse 会负责初始化此视图。一个视角可以包含多个视图，一个视图也可以属于多个视角。

15.2.2 “org.eclipse.ui.perspectiveExtensions”扩展点

在 Eclipse 中，用户可以建立自己的视角，把感兴趣的视图展现在自己的视角中，也可以把自己的视图放在已经建立的视角中。“org.eclipse.ui.perspectiveExtensions”为用户提供了把自己的视图放入相应视角的扩展。

例如用户有这样的场景：当切换 Eclipse 的 Resource 视角后要显示用户的视图，如图 15-2

所示。

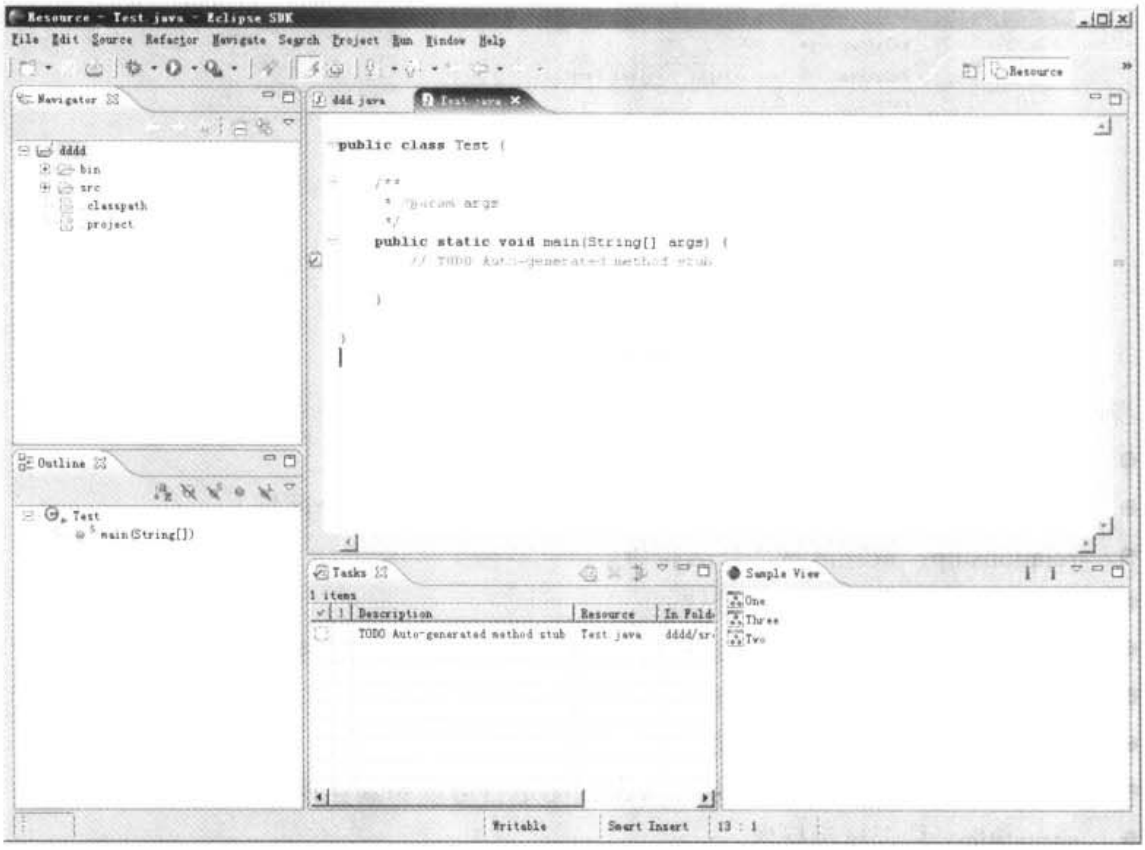


图 15-2　Resource 视角

当切换到 Resource 视角后，Eclipse 会自动把“Sample View”视图放在相应的位置。这就要求实现“org.eclipse.ui.perspectiveExtensions”扩展点。

“org.eclipse.ui.perspectiveExtensions”扩展点没有实现类，此扩展点只是作为一个描述，告诉 Eclipse 怎样对 Eclipse 中的相应视角进行布局。此扩展点有几个比较重要的描述信息：一是目标视角的 ID 号，表示要对哪一个视角的布局进行重定义；还要指定视图的 ID，表示把哪一个视图的引用放入目标视角；另外要指定视图在视角中的相应位置。

建立一个“org.eclipse.ui.perspectiveExtensions”扩展的步骤如下。

（1）在 Extensions 页中添加一个“org.eclipse.ui.perspectiveExtensions”的扩展。

（2）在“org.eclipse.ui.perspectiveExtensions”节点中添加一个“perspectiveExtension”节点，并指定它的 targetID 属性为“org.eclipse.ui.resourcePerspective”（表示 Resource 视角的 ID），如图 15-3 所示。

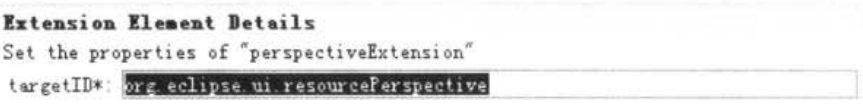


图 15-3　目标视角

（3）在“perspectiveExtension”节点下建立一个 view 节点，并设置好 view 节点的属性，通过这些属性可以设置指定视图在指定视角中的位置，如图 15-4 所示。

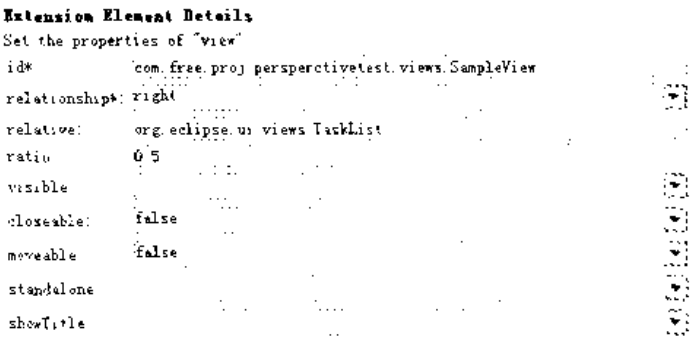


图 15-4 视图节点属性

视图节点属性解释如下。

- id：表示放入目标视角的视图 ID 号。
- relative：相关视图的 ID 号（即和哪一个视图放在同一个区域）。
- relationship：放在区域的什么位置。
- ratio：视图在区域中的百分比。
- visible：如果视图没有被初始化，第一次打开视角是否初始化此视图并显示。
- closeable：视图是否可关闭。
- moveable：视图是否可移动。
- standalone：是否是孤立的视图（即视图是否可以和其他视图放在一起）。
- showtitle：是否显示标题。

“perspectiveExtension”扩展点相当于起到把视角和视图 ID 绑定的作用，这样，Eclipse 会通过视角找到视图的引用。这种手法在 Eclipse 中是比较常见的。本例源程序可参考本书附带光盘，扩展点定义文件如例程 15-1 所示。

例程 15-1 plugin.xml

```
<plugin>
…
<extension
       point="org.eclipse.ui.perspectiveExtensions">
    <perspectiveExtension
        targetID="org.eclipse.ui.resourcePerspective">
      <view
          closeable="false"
          id="com.free.proj.persperctivetest.views.SampleView"
          moveable="false"
          ratio="0.5"
          relationship="right"
          relative="org.eclipse.ui.views.TaskList"
          visible="false">
      </view>
    </perspectiveExtension>
</extension>
…
</plugin>
```



用户不但可以把视图放到相应的视角，也可以把“ActionSet”之类的扩展放入相应的视角。

15.2.3　“org.eclipse.ui.perspectives”扩展点

上一节介绍了如何把自己的视图放入已有的视角中，但是用户如果想建立自己的视角应该如何实现呢？这就要求用户实现“org.eclipse.ui.perspectives”扩展点，步骤如下。

（1）在清单文件的 Extensions 页添加“org.eclipse.ui.perspectives”节点。

（2）在“org.eclipse.ui.perspectives”节点建立 perspective 节点，并设置相应的属性，如图 15-5 所示。

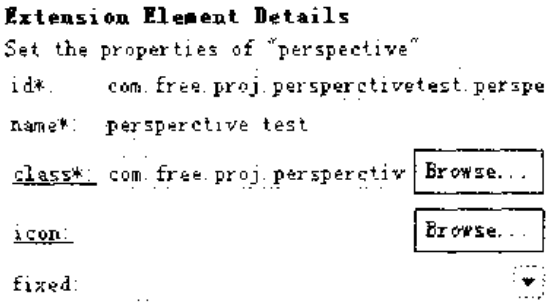


图 15-5　perspertive 节点属性

perspertive 节点属性解释如下。

- id：视角的 ID 号。
- name：视角的名称。
- class：视角扩展点的实现类。
- icon：视角的图标。

图 15-5 中视角扩展点的描述信息如下。

```
<extension
   point="org.eclipse.ui.perspectives">
 <perspective
     class="com.free.proj.persperctivetest.PerspectiveFactory"
     id="com.free.proj.persperctivetest.perspective"
     name="persperctive test"/>
</extension>
```

在此视角中，实现类为“PerspectiveFactory”。视角的实现类要求实现“IPerspectiveFactory”接口，通过 createInitialLayout(IPageLayout layout)方法实现视角中视图的布局。用户可以通过 layout 参数添加视图到当前的视角中，如例程 15-2 所示，添加了一个 ID 为“com.free.proj.persperctivetest.views.SampleView”的视图到当前视角。

例程 15-2　PerspectiveFactory.java

```
package com.free.proj.persperctivetest;
```

```
import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;

public class PerspectiveFactory implements IPerspectiveFactory {

public void createInitialLayout(IPageLayout layout) {
   layout.setFixed(true);
   layout.addView("com.free.proj.persperctivetest.views.SampleView",
               IPageLayout.LEFT, 0.5f,
               IPageLayout.ID_PROBLEM_VIEW);
   layout.setEditorAreaVisible(true);
  }}
```

IPageLayout 的 addView 方法定义为“public void addView(String viewId, int relationship, float ratio, String refId);”，参数解释如下。

- viewId：被添加到视角中的视图 ID 号。
- relationship：放在区域的什么位置。
- ratio：视图在区域中的百分比。
- refId：相关视图的 ID 号（即和哪一个视图放在同一个区域）。

打开选择视角的对话框，可以看到“persperctive test”视角，选择“persperctive test”视角，如图 15-6 所示。

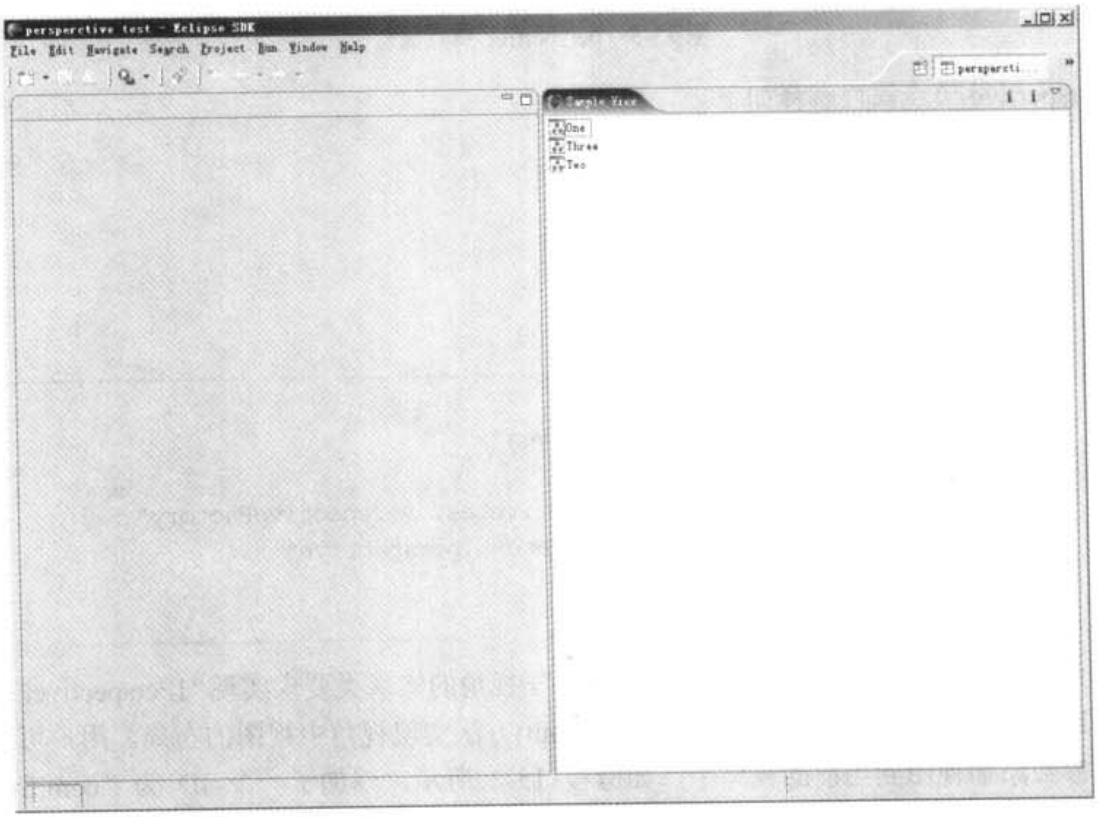


图 15-6 “persperctive test”视角

上例创建了用户自定义的视角，在视角中只添加了一个视图，如果在一些比较大的项目中，可能还会用到很多的视图，以及 Eclipse 原有的视图。用户也可以把这些视图添加到自定义的视角中，只需要知道这些视图的 ID 即可。

15.3　菜单工具栏

在插件入门这一章节，读者了解到如何用向导开发一个“Hello, world”插件。“Hello, world”插件用到的扩展点就是“org.eclipse.ui.actionSets”。本节就是要通过“org.eclipse.ui.actionSets”扩展点建立用户的菜单和工具栏。

15.3.1　“org.eclipse.ui.actionSets”扩展点信息

在 Eclipse 中，通过“org.eclipse.ui.actionSets”扩展点能够建立自己的菜单和工具栏。实现 ActionSet 扩展点的步骤如下。

（1）在插件清单文件的 Extensions 页上添加“org.eclipse.ui.actionSets”节点。

（2）在“org.eclipse.ui.actionSets”节点下添加 ActionSet 子节点。

（3）在 ActionSet 节点下添加 menu 和 action 子节点，如图 15-7 所示。

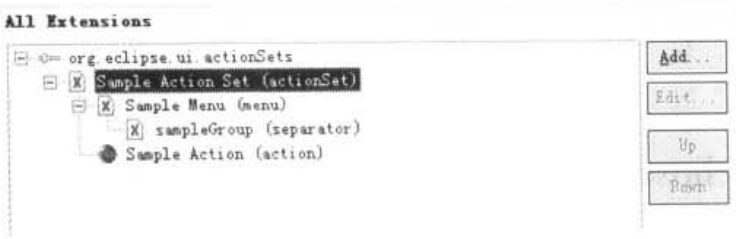


图 15-7　ActionSets 扩展点

menu 节点代表 Eclipse 的菜单项，menu 节点的属性设置如下。

- id：sampleMenu，id 表示 menu 的编号。
- label：Sample &Menu，label 表示在菜单中的显示名称。

action 节点代表菜单的菜单项或工具栏中按钮的相应动作，action 节点的属性设置如下。

- id：com.free.proj.actionset.actions.SampleAction，代表 Action 的唯一编号。
- label：&Sample Action，Action 的名称。
- menuBarPath：sampleMenu/sampleGroup，表示 Action 对应菜单的路径。
- toolbarPath：sampleGroup，表示 Action 对应工具栏按钮的路径。
- icon：icons/sample.gif，表示 Action 对应的图标。
- tooltip：Action 的提示信息。
- class：com.free.proj.actionset.actions.SampleAction，指定 Action 的实现类。

plugin.xml 文件如例程 15-3 所示。

例程 15-3　plugin.xml

```
<plugin>

  <extension
      point="org.eclipse.ui.actionSets">
    <actionSet
        label="Sample Action Set"
```

```
            visible="true"
            id="com.free.proj.actionset.actionSet">
          <menu
              id="sampleMenu"
              label="Sample &amp;Menu">
            <separator
                name="sampleGroup">
            </separator>
          </menu>
          <action
              label="&amp;Sample Action"
              icon="icons/sample.gif"
              class="com.free.proj.actionset.actions.SampleAction"
              tooltip="ActionSet Test"
              menubarPath="sampleMenu/sampleGroup"
              toolbarPath="sampleGroup"
              id="com.free.proj.actionset.actions.SampleAction">
          </action>
        </actionSet>
      </extension>

    </plugin>
```

15.3.2 “org.eclipse.ui.actionSets”扩展点的实现类“SampleAction”

从 plugin.xml 文件可以看出 Action 对应的实现类为 SampleAction。SampleAction 实现了 IWorkbenchWindowActionDelegate 接口，其中 run(IAction action)方法为用户单击菜单或工具栏按钮时的响应方法，SampleAction 代码如例程 15-4 所示。

例程 15-4　SampleAction.java

```
package com.free.proj.actionset.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.dialogs.MessageDialog;

public class SampleAction implements IWorkbenchWindowActionDelegate {
    private IWorkbenchWindow window;
    public SampleAction() {
    }

    public void run(IAction action) {
        MessageDialog.openInformation(
            window.getShell(),
            "Actionset Plug-in",
            "ActionSet Test");
    }
```

```
    public void selectionChanged(IAction action, ISelection selection) {
    }

    public void dispose() {
    }

    public void init(IWorkbenchWindow window) {
        this.window = window;
    }
}
```

上例中，通过 run 方法响应用户单击菜单和工具栏的请求。在 Eclipse 中，Action 是可以被菜单和工具栏所复用的，程序运行效果可参考插件入门章节的实例。

15.4　项目相关特性

本节将从“Hello world, project ”具体的实例出发，在 Eclipse 项目中添加用户自己的项目特性，扩展 Eclipse 的选项，让用户的插件具有一个产品的雏形。

15.4.1　项目扩展点

前面的章节，用户已经了解了如何在 Eclipse 中添加视图、编辑器、视角菜单和工具栏按钮。本节将要介绍如何加入自己的项目相关特性，其中包括向导、项目特性、构建器和属性页。项目特性扩展点如图 15-8 所示。

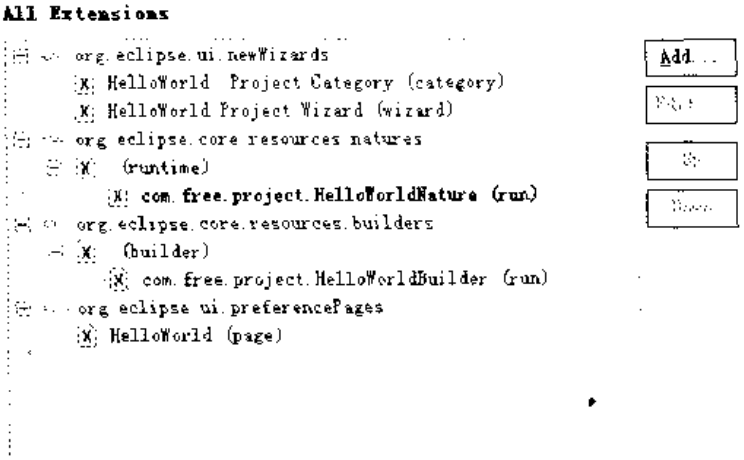


图 15-8　项目特性扩展点

15.4.2　“org.eclipse.core.resources.natures”扩展点及实现类

nature 为用户添加项目的特有特性，Eclipse 中一个项目可以同时具有多个 nature 特性。例如用户要在 Java 项目的基础上扩展自己的项目，加入了自己项目的特有特性。这样 JDT 就能认识用户新建的项目，用户也可以判断此 Java 项目是否是自己的扩展。一般来说，用户期望

用到 JDT 的强大功能，就要使自己的项目从 Java 项目扩展过来。

在插件中添加 nature 扩展点，步骤如下。

（1）在插件清单文件的 Extensions 页中添加“org.eclipse.core.resources.natures”扩展点，设置 ID 属性为“helloworldnature”。

（2）在“org.eclipse.core.resources.natures”节点下添加 runtime 子节点。

（3）在 runtime 节点下添加 run 节点，设置实现类 class 为“com.free.project.HelloWorldNature”。

扩展点在 plugin.xml 文件中的描述如下。

```
<extension
      id="helloworldnature"
      point="org.eclipse.core.resources.natures">
   <runtime>
      <run class="com.free.project.HelloWorldNature"/>
   </runtime>
</extension>
```

nature 扩展点的实现类要实现“IProjectNature”接口，并要实现“IProjectNature”的如下几个方法。

- setProject(IProject project)：在 nature 类中保存当前项目。
- getProject()：从 nature 中得到当前项目。
- configure()：当 nature 被添加到项目中时，通过 configure 方法配置当前项目。
- deconfigure()：当 nature 从项目删除时，通过 deconfigure 方法删除相应配置。

nature 对应的实现类为“HelloWorldNature”，在此实现类中主要功能是通过 configure 方法配置当前项目的构建器，代码如例程 15-5 所示。

例程 15-5　HelloWorldNature.java

```
package com.free.project;

import org.eclipse.core.resources.ICommand;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IProjectDescription;
import org.eclipse.core.resources.IProjectNature;
import org.eclipse.core.runtime.CoreException;

public class HelloWorldNature implements IProjectNature {

   private IProject _project;
   //配置Nature
   public void configure() throws CoreException {
    System.out.println("configure");
      IProjectDescription projectDesc = _project.getDescription();
      ICommand[] buildSpec = projectDesc.getBuildSpec();
      boolean hasBuilder = false;
      //遍历项目的构建器
      for (int i = 0; i < buildSpec.length; ++i) {
            if (buildSpec[i]
```

```
                .getBuilderName()
                .equals("com.free.proj.helloproject.helloworldbuilder")) {
                hasBuilder = true;
                System.out.println("true");
                break;
            }
        }

        if (hasBuilder == false) {
            System.out.println("false");
            ICommand newCommand = projectDesc.newCommand();
            newCommand.setBuilderName("com.free.proj.helloproject.helloworldbuilder");
            ICommand[] buildSpecs = new ICommand[buildSpec.length + 1];

            System.arraycopy(buildSpec, 0, buildSpecs, 1, buildSpec.length);
            buildSpecs[0] = newCommand;
            projectDesc.setBuildSpec(buildSpecs);
            _project.setDescription(projectDesc, null);
        }
    }

    public void deconfigure() throws CoreException {

    }
    //获得Nature所关联的项目
    public IProject getProject() {
     System.out.println("getProject");
            return _project;
    }
    //设置Nature所关联的项目
    public void setProject(IProject project) {
      System.out.println("setProject");
      _project = project;
    }

}
```

注意 此外构建器的ID号为“com.free.proj.helloproject.helloworldbuilder”，ID号为插件ID号加构建器ID号，在Eclipse中和产品及项目相关的ID都采用这种方式。

15.4.3 “org.eclipse.core.resources.builders”扩展点及实现类

Builder为项目的构建器，通过构建器能实现项目的增量构建和完全构建。在插件中添加Builder扩展点，步骤如下。

（1）在插件清单文件的Extensions页中添加“org.eclipse.core.resources.builders”扩展点，设置ID属性为“helloworldbuilder”，name属性为“hello world, bulder”。

（2）在“org.eclipse.core.resources.builders”节点下添加builder子节点。

（3）在 builder 节点下添加 run 节点，设置实现类 class 为“com.free.project.HelloWorldBuilder”。

扩展点在 plugin.xml 文件中的描述如下。

```
<extension
      id="helloworldbuilder" name="hello world, bulder"
      point="org.eclipse.core.resources.builders">
    <builder>
      <run class="com.free.project.HelloWorldBuilder"/>
    </builder>
  </extension>
```

builder 扩展点的实现类继承于“IncrementalProjectBuilder”类，并实现了两个资源监听器。当项目中资源发生变化的时候调用构建，本例中只打印出相应的数据，并没有实现构建的动作，当用户需要构建时可以替换相应代码，例如编译文件等。代码如例程 15-6 所示。

例程 15-6　HelloWorldBuilder.java

```
package com.free.project;

import java.util.Map;

import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.resources.IResourceDelta;
import org.eclipse.core.resources.IResourceDeltaVisitor;
import org.eclipse.core.resources.IResourceVisitor;
import org.eclipse.core.resources.IncrementalProjectBuilder;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;

public class HelloWorldBuilder extends IncrementalProjectBuilder {
   //实现资源监听器
   private class HelloWorldVisitor implements IResourceVisitor {
      public boolean visit(IResource resource) throws CoreException {
         System.out.println("**** HelloWorldVisitor.visit() ****");
         switch (resource.getType()) {
         case IResource.PROJECT:
            System.out.println("Project added: " + resource.getName());
            break;
         case IResource.FOLDER:
            System.out.println("Folder added: " + resource.getName());
            break;
         case IResource.FILE:
            System.out.println("File added: " + resource.getName());
            break;
         }

         return true;
      }
   }
```

```
//实现资源监听器
private class HelloWorldDeltaVisitor implements IResourceDeltaVisitor {
   public boolean visit(IResourceDelta delta) throws CoreException {
      System.out.println("**** HelloWorldDeltaVisitor.visit() ****");
      String type = null;
      switch (delta.getResource().getType()) {
      case IResource.ROOT:
         type = "ROOT";
         break;
      case IResource.PROJECT:
         type = "Project";
         break;
      case IResource.FOLDER:
         type = "Folder";
         break;
      case IResource.FILE:
         type = "File";
         break;
      }

      switch (delta.getKind()) {
      case IResourceDelta.ADDED:
         System.out.println(type + " added: "+ delta.getResource().getName());
         break;
      case IResourceDelta.CHANGED:
         System.out.println(type + " changed: "
               + delta.getResource().getName());
         break;
      case IResourceDelta.REMOVED:
         System.out.println(type + " removed: "
               + delta.getResource().getName());
         break;
      }

      return true;
   }
}

public HelloWorldBuilder() {
   System.out.println("HelloWorldBuilder.constructor()");
}
//开始构建
protected IProject[] build(int kind, Map args, IProgressMonitor monitor)
      throws CoreException {
   System.out.println("HelloWorldBuilder.build()");
   //当构建类型不同时，添加不同的资源监听器
   if (kind == IncrementalProjectBuilder.FULL_BUILD) {
      System.out.println("FULL_BUILD");
      getProject().accept(new HelloWorldVisitor());
   } else {
      IResourceDelta delta = getDelta(getProject());
```

```
            if (delta == null) {
                System.out.println("AUTO_BUILD");
                getProject().accept(new HelloWorldVisitor());
            } else {
                System.out.println("INCREMENTAL_BUILD");
                delta.accept(new HelloWorldDeltaVisitor());
            }
        }
        return null;
    }

}
```

在用户建立的项目中，如果资源发生改变，builder 会调用“HelloWorldDeltaVisitor”类进行增量构建处理，当清除项目时会调用“HelloWorldVisitor”类进行完全构建处理。插件运行后，当用户选择自定义项目时，用户可以打开项目属性窗口看到“hello world, builder”构建器，如图 15-9 所示。

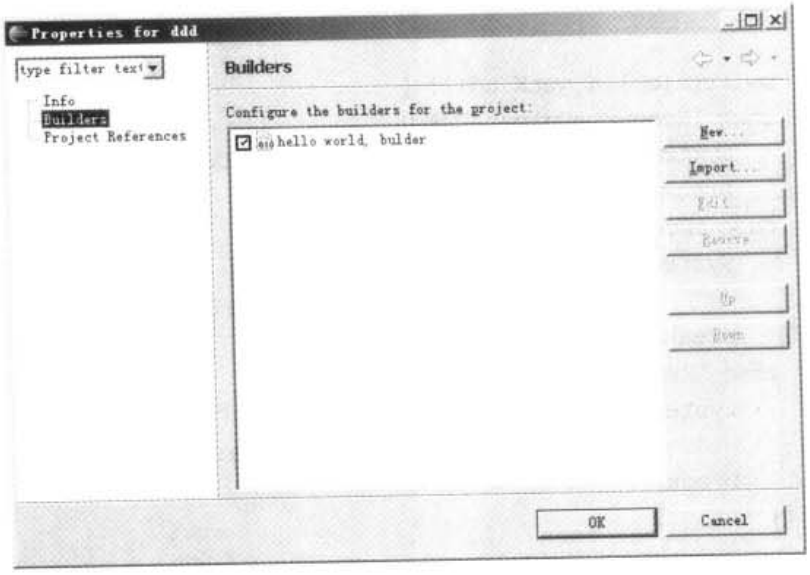


图 15-9　项目属性窗口

15.4.4　“org.eclipse.ui.newWizards”扩展点及实现类

用户如果要在项目中添加自定义的资源，采用 Eclipse 中的向导是最好的方式。项目也是资源的一种，一般来说如果对新建的资源有输入条件的要求，都采用向导的方式。例如新建 Java 项目，要输入项目名称和构建路径等。在插件中添加 newWizard 扩展点，步骤如下。

（1）在插件清单文件的 Extensions 页中添加“org.eclipse.ui.newWizards”扩展点。

（2）在“org.eclipse.ui.newWizards”节点下添加 category 子节点，设置 ID 属性为“com.free.project.wizard.category”，name 属性为“HelloWorld Project Category”。

（3）在“org.eclipse.ui.newWizards”节点下添加 wizard 节点，设置实现类 class 为“com.free.project.HelloWorldWizard”，ID 属性为“com.free.project.wizard”。

扩展点在 plugin.xml 文件中的描述如下：

```
<extension
      point="org.eclipse.ui.newWizards">
    <category
        name="HelloWorld  Project Category"
        id="com.free.project.wizard.category"/>
    <wizard
        class="com.free.project.HelloWorldWizard"
        category="com.free.project.wizard.category"
        name="HelloWorld Project Wizard"
        id="com.free.project.wizard"/>
</extension>
```

newWizards 扩展点的实现类继承于“org.eclipse.jface.wizard.Wizard”类，并实现了“org.eclipse.ui.INewWizard”接口。在实现类中，通过 addPages 方法添加向导页到当前向导，通过 performFinish 方法完成向导的结束。实现类“HelloWorldWizard”的代码如例程 15-7 所示。

例程 15-7　HelloWorldWizard.java

```
package com.free.project;

import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IProjectDescription;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.wizard.Wizard;
import org.eclipse.ui.INewWizard;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.dialogs.WizardNewProjectCreationPage;

public class HelloWorldWizard extends Wizard implements INewWizard {

   private WizardNewProjectCreationPage _page;

   public HelloWorldWizard() {
      super();
   }
   //添加向导页
   public void addPages() {
      _page = new WizardNewProjectCreationPage("page1");
      _page.setDescription("Page 1 of Helloworld New Wizard");
      _page.setTitle("HelloWorld Wizard");
      addPage(_page);
   }
   //完成向导
   public boolean performFinish() {
      boolean result = false;

      System.out.println("project page.name = " + _page.getProjectName());
      System.out.println("project page.path = " + _page.getLocationPath());
      IProject project = _page.getProjectHandle();
```

```
            try {
                project.create(null);
                result = true;
            } catch (CoreException e) {
                e.printStackTrace();
            }

            IProjectDescription projectDesc = null;
            try {
                project.open(null);
                projectDesc = project.getDescription();
            } catch (CoreException e1) {
                e1.printStackTrace();
            }

            String[] natureIds = projectDesc.getNatureIds();
            String[] newNatureIds = new String[natureIds.length + 1];
            System.arraycopy(natureIds, 0, newNatureIds, 0, natureIds.length);
            newNatureIds[natureIds.length] = "com.free.proj.helloproject.helloworldnature";
            projectDesc.setNatureIds(newNatureIds);
            try {
                project.setDescription(projectDesc, null);
            } catch (CoreException e2) {
                e2.printStackTrace();
            }

            return result;
        }

        public void init(IWorkbench workbench, IStructuredSelection selection) {

        }

    }
```

本例程中没有实现用户定义的向导页，而是用 Eclipse 下的“WizardNewProjectCreationPage”页来作为向导页。用户也可以定义向导页，如果有兴趣可以参考多页签编辑器中的例子源代码，多页签编辑器插件已经实现了向导。本例中可以通过新建菜单新建用户定义的项目，向导对话框如图 15-10 所示。

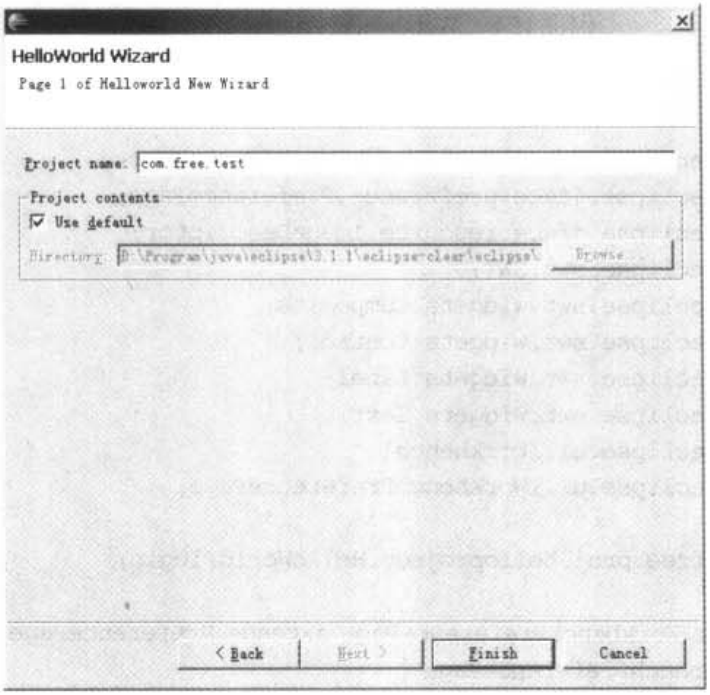


图 15-10　向导对话框

15.4.5　“org.eclipse.ui.preferencePages”扩展点及实现类

在用户定义的项目中，一般都会有相应的参数配制。例如说 Tomcat 要配置相应的 Web 服务器，JDT 可以配置代码的风格、JDK 和 JRE 等。在插件中可以添加 preferencePages 扩展点，实现项目参数的配置，步骤如下。

（1）在插件清单文件的 Extensions 页中添加“org.eclipse.ui.preferencePages”扩展点。

（2）在“org.eclipse.ui.preferencePages”节点下添加 page 子节点，设置节点的 ID 属性为“com.free.project.page”，name 属性为“HelloWorld”，class 属性为“com.free.project.WorkbenchPreferencePage”。

扩展点在 plugin.xml 文件中的描述如下。

```
<extension
   point="org.eclipse.ui.preferencePages">
 <page
      class="com.free.project.WorkbenchPreferencePage"
      name="HelloWorld"
      id="com.free.project.page"/>
</extension>
```

preferencePages 扩展点的实现类继承于“org.eclipse.jface.preference.PreferencePage”类，并实现了“org.eclipse.ui.IWorkbenchPreferencePage”接口。在实现类中，通过 createContents 方法构建自己的属性页，通过 performOk 方法完成向导的结束。实现类“WorkbenchPreferencePage”的代码如例程 15-8 所示。

例程 15-8 WorkbenchPreferencePage.java

```
package com.free.project;

import org.eclipse.jface.preference.IPreferenceStore;
import org.eclipse.jface.preference.PreferencePage;
import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPreferencePage;

import com.free.proj.helloproject.HelloWorldPlugin;

public class WorkbenchPreferencePage extends PreferencePage implements
      IWorkbenchPreferencePage {

   private Text _greeting;

   public WorkbenchPreferencePage() {
      super();
   }

   public WorkbenchPreferencePage(String title) {
      super(title);
   }

   public WorkbenchPreferencePage(String title, ImageDescriptor image) {
      super(title, image);
   }

   protected Control createContents(Composite parent) {
      Label label = new Label(parent, SWT.CENTER);
      label.setText("Greeting");
      _greeting = new Text(parent, SWT.SINGLE | SWT.BORDER);

      return parent;
   }

   protected IPreferenceStore doGetPreferenceStore() {
      System.out.println("WorkbenchPreferencePage.doGetPreferenceStore()");
      return HelloWorldPlugin.getDefault().getPreferenceStore();
   }

   public void init(IWorkbench workbench) {

   }
```

```
    protected void performDefaults() {
        System.out.println("WorkbenchPreferencePage.performDefaults()");
        IPreferenceStore prefStore = getPreferenceStore();
        prefStore.setValue(HelloWorldPlugin.GREETING, "world");
        HelloWorldPlugin.getDefault().savePluginPreferences();

        _greeting.setText(prefStore.getString(HelloWorldPlugin.GREETING));
    }

    public boolean performOk() {
        boolean result = false;

        System.out.println("WorkbenchPreferencePage.performOk()");
        IPreferenceStore prefStore = getPreferenceStore();
        prefStore.setValue(HelloWorldPlugin.GREETING, _greeting.getText());
        HelloWorldPlugin.getDefault().savePluginPreferences();

        result = true;

        return result;
    }
}
```

本例程只添加了一个 Label 控件和一个 Text 控件。若用户有兴趣可以添加自己完成的属性页，选择“window”→“preferences”菜单，打开首选项对话框，程序运行效果如图 15-11 所示。

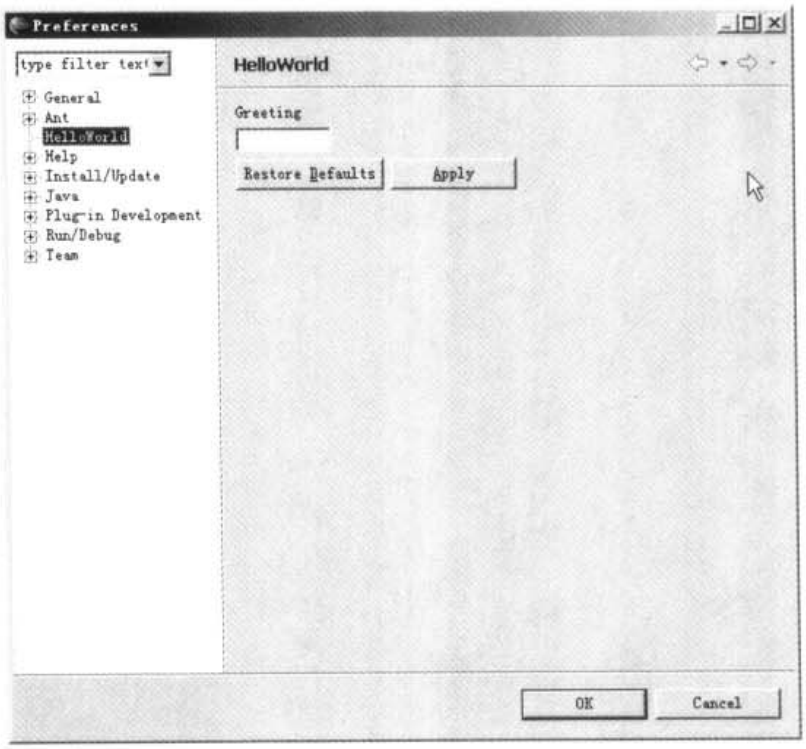


图 15-11　首选项窗口

15.5 小　　结

本章介绍了项目中常用扩展点的实现方式，以及通过实例建立用户自定义的项目。通过对本章的学习，读者应该了解视角和项目相关扩展点的实现原理。有些概念对初学者来说可能比较难懂，例如 natures 和 builders，希望读者可以参考 JDT 中项目特性和构建的方式。

第 16 章　帮助相关扩展点的实现

在产品开发中，帮助文档是比较重要的一个部分，它为用户使用产品提供了重要的信息。本章将通几个例子较全面地剖析如何开发 Eclipse 中的帮助文档。

本章主要从以下几个部分进行介绍。

- 联机帮助
- 欢迎界面帮助
- 上下文帮助

16.1　联 机 帮 助

在 Eclipse 中，联机帮助是使用者获得帮助信息的主要手段。用户可以通过“Help”→“Help Contents”菜单打开联机帮助文档。

Eclipse 的联机帮助文档可以包含大量的信息，帮助文档的内容是以 HTML 文件的形式组织的，帮助文档的结构是以 XML 文件的形式进行组织的。也就是说，用户既可以复用已有的 HTML 文件形式的帮助，又可以利用 XML 文件结构化地组织帮助。

16.1.1　联机帮助扩展点的实现

在 Eclipse 中，一切都是插件，如果添加用户的联机帮助文档，用户需要实现 Eclipse 的“org.eclipse.help.toc”扩展点，添加“org.eclipse.help.toc”扩展点的步骤如下。

（1）在清单文件的 Extensions 页添加“org.eclipse.ui.toc”节点。

（2）在“org.eclipse.help.toc”节点上添加若干个 toc 子节点，如图 16-1 所示。

设置 toc 子节点的 file 属性为相应的 XML 文件，其中有一个节点的 primary 值为 true，说明此节点为帮助文档的入口节点，如图 16-2 所示。

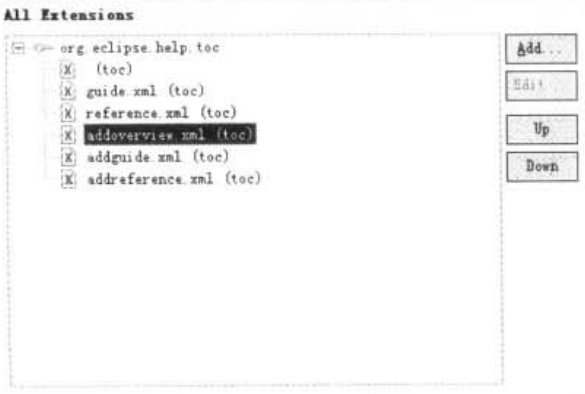


图 16-1　toc 节点结构

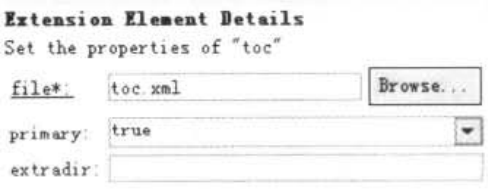


图 16-2　toc 子节点属性

另外 toc 子节点提供了 XML 文件在 Eclipse 中的注册信息，联机帮助插件的 plugin.xml 文件如例程 16-1 所示。

例程 16-1　plugin.xml

```
<plugin>

  <extension point="org.eclipse.help.toc">
    <toc file="toc.xml" primary="true"/>
    <toc file="guide.xml"/>
    <toc file="reference.xml"/>
    <toc file="addoverview.xml"/>
    <toc file="addguide.xml"/>
    <toc file="addreference.xml"/>
  </extension>

</plugin>
```

从 plugin.xml 文件可以看出每个 toc 子节点都对应一个 XML 文件，通过 XML 文件可以组织帮助子节点的结构。

16.1.2　联机帮助文档的结构

Eclipse 的联机帮助文档不需要有扩展点的实现类，联机帮助文档是通过 XML 文件进行组织的。如插件的清单文件所示，在扩展点中只注册了相关的 XML 文件，这些 XML 文件之间又会相互引用，从而组成一个以 XML 文件为基础的树型结构。

Eclipse 提供多种场景帮助文档的实现。

场景一：用户想一次性完成帮助文档。在这种情况下，用户可以直接构建自己的 XML 树。

场景二：用户不能够一次性完成帮助文档。在这种情况下，用户可以在 XML 文档中预置锚点，其他用户想实现这方面的帮助时，把自己的帮助挂接到此锚点即可（也是就说，用户可以通过这种实现方式把自己的帮助挂接在 JDT 的帮助文档中）。

下面将给出这两种场景下 XML 文档的实现。

从 plugin.xml 文件可以看出，帮助文档入口文件为 toc.xml，此文件展示了帮助文档的第一层节点。XML 文件如例程 16-2 所示。

例程 16-2　toc.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<toc label="Free Help of Contents" topic="help/titlepage.html">
   <topic label="Overview" href="help/overview.html"/>

   <topic label="Guide sub-section" >
      <link toc="guide.xml"/>
      <anchor id="afterGuide"/>
   </topic>

   <topic label="Reference sub-section">
```

```
        <link toc="reference.xml"/>
        <anchor id="afterReference"/>
      </topic>
    </toc>
```

帮助文档入口文件中定义第一层节点有 3 个，下面分别解释这 3 个节点。

- 第一个节点是叶子节点，用 href 标记链接到“help/overview.html”文件。
- 第二个节点是子节点，子节点下有一个节点通过标记“link”到 guide.xml，通过 guide.xml 进行定义，另外一个子节点通过“id”定义了一个锚点“afterGuide”，用户可以把自己的 XML 文件挂接到此锚点。
- 第三个节点结构和第二个节点一样。

通过“link”方式连接的 XML 文件的结构和 toc.xml 文件类似，例如 guide.xml 的定义如例程 16-3 所示。

例程 16-3　guide.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<toc label="Guide sub-section" topic="help/guide.html">
   <topic label="Guide" href="help/guide1.html"/>
</toc>
```

通过锚点挂接的 XML 文件也只能多出一个“link_to”的标记挂接到相应锚点，定义如例程 16-4 所示。

例程 16-4　addguide.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<toc label="Additional Guide" link_to="toc.xml#afterGuide" >
   <topic label="Additional Guide 1" href="help/anchor2.html"/>
</toc>
```

通过扩展点的定义和相应节点的 XML 文件定义，Eclipse 就会构造出一个联机帮助文档的树，树的叶子节点会链接到相应的 HTML 文件，如图 16-3 所示。



图 16-3　联机帮助文档

16.1.3 在联机帮助中在线调用 Java 类

联机帮助文档是纯文本的文件，不能够编译运行。但有些时候，用户可能需要在帮助文档中演示只有在 Java 代码中才能完成的操作，Eclipse 也提供这方面的支持，用户可以在帮助文档的页面中调用相应的 Java 类。

如果要在网页中调用 Java 类，在用户的网页下要引入“org.eclipse.help/livehelp.js”文件，通过此 .js 文件调用相应的 Java 代码，实现代码如例程 16-5 所示。

例程 16-5 HelloHelp.html

```
<html>
 <head>
  <title>Example Active Help Page</title>
  <script language="JavaScript" src="../../org.eclipse.help/livehelp.js">
  </script>
 </head>
 <body>
  <h2>Active Help</h2>
  This text is too confusing to understand. Click
  <a href='javascript:liveAction(
   "com.free.help.activehelp",
   "com.free.help.activehelp.HelloWorldActiveHelp",
   "Kitty"
  )'>
  here</a> for more information.
 </body>
</html>
```

单击此页面的链接将会执行“com.free.help.activehelp”插件下的“com.free.help.activehelp.HelloWorldActiveHelp”文件，参数为“Kitty”，HelloWorldActiveHelp 实现了“ILiveHelpAction”接口，通过“setInitializationString(String data)”方法传递来自页面的参数，代码如例程 16-6 所示。

例程 16-6 HelloWorldActiveHelp.java

```
package com.free.help.activehelp;

import org.eclipse.help.ILiveHelpAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.PlatformUI;

public class HelloWorldActiveHelp implements ILiveHelpAction {

   private class HelloWorldInnerClass implements Runnable {

      private String _name;
```

```
        public HelloWorldInnerClass(String name) {
            _name = name;
        }

        public void run() {
            IWorkbenchWindow window =
                PlatformUI.getWorkbench().getActiveWorkbenchWindow();
            if (window != null) {
                Shell shell = window.getShell();
                shell.setMinimized(false);
                shell.forceActive();
                //打开一个通信对话框
                MessageDialog.openInformation(window.getShell(), "Hello, " + _name,
                "Hello, " + _name);
            }
        }
    }

    private String _name;

    public void setInitializationString(String data) {
        _name = data;
    }
    public void run() {
        Display.getDefault().syncExec(new HelloWorldInnerClass(_name));
    }
}
```

当用户单击帮助文档的链接时，将会执行关联的 Java 类，程序执行效果如图 16-4 所示。

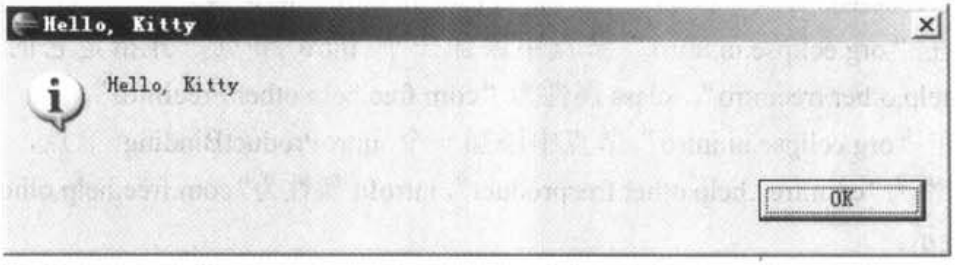


图 16-4　页面调用 Java 代码

16.2　欢迎界面帮助

当用户第一次运行 Eclipse 后，首先看到的是一个欢迎界面，用户也可以通过“Help”→“Welcome”菜单打开欢迎界面。欢迎界面介绍了产品的内容及教程等信息，对初学者来说会有比较大的帮助。如果用户要开发 Eclipse 下的一个产品，欢迎界面是必备的。Eclipse 提供了对欢迎界面的扩展。

16.2.1 产品扩展点

欢迎界面扩展点一般要求和产品扩展点绑定，如果用户想把自己的产品集成到 Eclipse 中，并且有自己的欢迎界面，就有必要先实现产品的扩展点，然后把欢迎界面绑定到产品中。在介绍欢迎界面扩展点之前先来看看产品扩展点。添加一个产品扩展点的步骤如下。

（1）在 Extensions 页中添加一个“org.eclipse.core.runtime.products”的扩展，设置产品的 ID 为“freeproduct”。

（2）在“org.eclipse.core.runtime.products”节点中添加一个“product”节点，并指定它的 application 属性为“org.eclipse.ui.ide.workbench”，name 属性为“free app”，如图 16-5 所示。

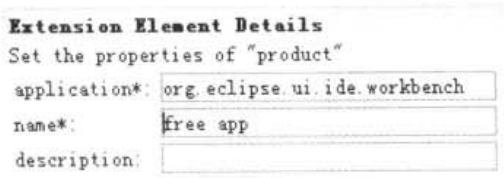


图 16-5 产品属性

产品扩展点没有实现类，但它可以绑定应用（即平台可运行的程序），图 16-5 所示的 application 就是产品绑定的应用，本例中没有实现自己的应用，通过在产品中引用 Eclipse 的“org.eclipse.ui.ide.workbench”应用运行。

16.2.2 欢迎界面扩展点

如果用户在运行时指定特定的产品，Eclipse 在启动后会打开用户定义的欢迎界面，添加欢迎界面扩展点的步骤如下。

（1）在 Extensions 页中添加一个“org.eclipse.ui.intro”的扩展。

（2）在“org.eclipse.ui.intro”节点中添加一个 intro 节点，并指定它的 id 属性为“com.free.help.other.freeintro”，class 属性为“com.free.help.other.FreeIntro”。

（3）在“org.eclipse.ui.intro”节点中添加一个 introProductBinding 节点，并指定它的 productId 属性为“com.free.help.other.freeproduct”，introId 属性为“com.free.help.other.freeintro”，如图 16-6 所示。

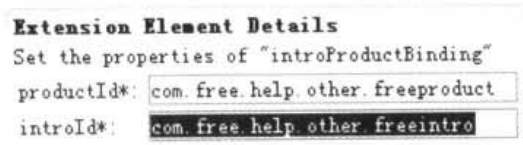


图 16-6 introProductBinding 属性

introProductBinding 负责把欢迎界面和产品绑定起来，当用户运行此产品时，调用绑定的欢迎界面。

16.2.3　插件清单文件

插件清单文件定义了产品和欢迎界面的扩展点。产品和欢迎界面的插件清单文件如例程 16-7 所示。

例程 16-7　plugin.xml

```
<plugin>
  <extension id="freeproduct" point="org.eclipse.core.runtime.products">
    <product
        application="org.eclipse.ui.ide.workbench"
        name="free app"/>
  </extension>
  <extension
      point="org.eclipse.ui.intro">
    <intro
        class="com.free.help.other.FreeIntro"
        id="com.free.help.other.freeintro"/>
    <introProductBinding
        introId="com.free.help.other.freeintro"
        productId="com.free.help.other.freeproduct"/>
  </extension>

</plugin>
```

16.2.4　扩展点实现类

欢迎界面扩展点的实现类“FreeIntro”继承于 IntroPart 类，通过 createPartControl (Composite parent)方法建立用户的欢迎界面。可以看出，createPartControl 方法的参数为 SWT 的 Composite 组件，用户可以在此组件上使用 SWT 实现自己的欢迎界面，实现类代码如例程 16-8 所示。

例程 16-8　FreeIntro.java

```
package com.free.help.other;

import org.eclipse.jface.resource.JFaceResources;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.forms.HyperlinkSettings;
import org.eclipse.ui.forms.events.HyperlinkAdapter;
import org.eclipse.ui.forms.events.HyperlinkEvent;
import org.eclipse.ui.forms.widgets.Form;
import org.eclipse.ui.forms.widgets.FormText;
```

```
import org.eclipse.ui.forms.widgets.FormToolkit;
import org.eclipse.ui.forms.widgets.TableWrapLayout;
import org.eclipse.ui.intro.IIntroManager;
import org.eclipse.ui.part.IntroPart;

public class FreeIntro extends IntroPart {

    Form introForm;

    public void setFocus() {
        introForm.setFocus();
    }

    //建立用户自定义的欢迎界面
    public void createPartControl(Composite parent) {
        FormToolkit tk = new FormToolkit(parent.getDisplay());
        introForm = tk.createForm(parent);
        TableWrapLayout layout = new TableWrapLayout();
        introForm.getBody().setLayout(layout);
        FormText tx = tk.createFormText(introForm.getBody(), true);
        tx.setParagraphsSeparated(true);
        HyperlinkSettings settings = new HyperlinkSettings(parent.getDisplay());
        settings.setHyperlinkUnderlineMode(HyperlinkSettings.UNDERLINE_HOVER);
        tx.setHyperlinkSettings(settings);
        String text = "<form><p><span font=\"title\">free propuct test</span></p>"
                + "<p><span color=\"subtitle\" font=\"subtitle\">"
                + "go:</span></p>"
                + "<p><a href=\"http://getStart\">getStart</a></p></form>";
        tx.setText(text, true, false);
        Font titleFont = JFaceResources.getFont(JFaceResources.HEADER_FONT);
        tx.setFont("title", titleFont);
        Font subtitleFont = JFaceResources.getFont(JFaceResources.BANNER_FONT);
        tx.setFont("subtitle", subtitleFont);
        Color col = parent.getDisplay().getSystemColor(SWT.COLOR_DARK_GREEN);
        tx.setColor("subtitle", col);
        //添加链接的监听器
        tx.addHyperlinkListener(new HyperlinkAdapter() {
            public void linkActivated(HyperlinkEvent e) {
                // Fetch IntroManager, close welcome screen
                IWorkbenchWindow window = getIntroSite().getWorkbenchWindow();
                IWorkbench workbench = window.getWorkbench();
                IIntroManager manager = workbench.getIntroManager();
                manager.closeIntro(FreeIntro.this);
            }
        });
    }

    public void standbyStateChanged(boolean standby) {
    }
}
```

16.2.5　调试欢迎界面插件

此处的欢迎界面扩展点是和产品扩展点绑定的，要想调试用户定制的欢迎界面插件，就要在调试时选择用户定制的产品，步骤如下。

（1）打开菜单“Run”→“Debug”，打开 Debug 插件参数设置窗口。

（2）设置运行参数“Run a product”为用户定义产品的扩展点，如图 16-7 所示。

（3）单击“Debug”按钮，开始调试自定义的产品，欢迎界面运行效果如图 16-8 所示。

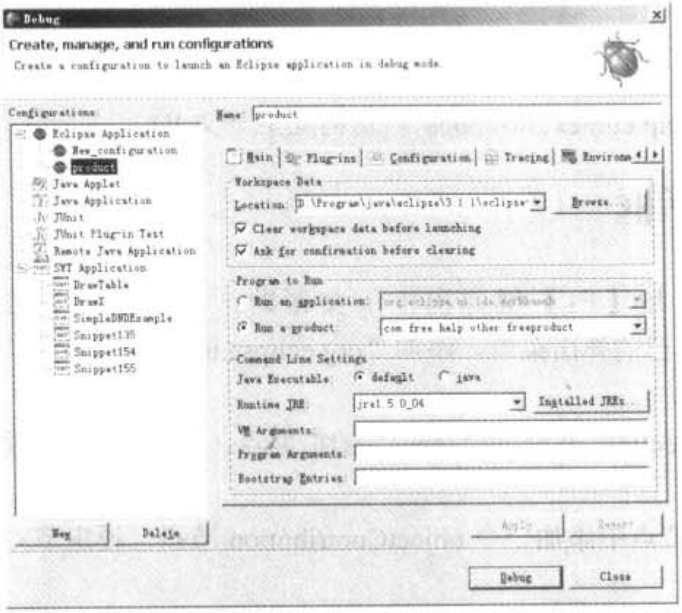


图 16-7　设置 Debug 参数



图 16-8　欢迎界面

当用户启动调试环境后，Eclipse 会启动产品所定义的应用，首先将会打开用户定义的欢迎界面。

16.3　上下文帮助

上下文帮助是根据用户当前的场景给用户提示的帮助。在 Eclipse 中，用户在某个场景按【F1】键就会弹出上下文帮助信息。

16.3.1　定义上下文帮助

用户如果要实现上下文帮助，首先需要添加上下文帮助的扩展点，步骤如下。

（1）在 Extensions 页中添加一个“org.eclipse.help.contexts”的扩展点。

（2）在“org.eclipse.help.contexts”节点中添加一个 contexts 节点，并指定它的 file 属性为“contexts_Confusing.xml”。

Eclipse 通过 XML 文件设定上下文帮助信息的结构，用 HTML 文件给出帮助内容。contexts_Confusing.xml 内容如例程 16-9 所示。

例程 16-9　contexts_Confusing.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<contexts>
    <context id="infopops">
        <description>Don't Panic!</description>
        <topic href="help/overview.html" label="Who is Deep Thought?"/>
        <topic href="help/guide.html" label="Deep Thought Handbook"/>
        <topic href="help/references.html" label="Deep Thought API"/>
    </context>
</contexts>
```

其中，contexts_Confusing.xml 设定了 ID 值为“infopops”，上下文 ID 全称为插件的 ID 加上下文 ID，此例中用户要通过“com.free.help.context.infopops” 访问此上下文 ID。

16.3.2　定义弹出菜单和相应对话框

本例中要在弹出菜单的相应窗口中按【F1】键弹出上下文帮助，用户可以通过“org.eclipse.ui.popupMenus”扩展点添加自己的弹出菜单。添加“org.eclipse.ui.popupMenus”扩展点的步骤如下。

（1）在 Extensions 页中添加一个“org.eclipse.ui.popupMenus”的扩展点，设置 name 属性为“Confusing Input Dialog”。

（2）在“org.eclipse.ui.popupMenus”节点中添加一个 objectContribution 节点，设定节点的相关属性，如图 16-9 所示。

nameFilter 属性为“*.gan”，objectClass 属性为“org.eclipse.core.resources.IFile”，说明用户只有在扩展名为 gan、对象类型为“org.eclipse.core.resources.IFile”的文件上单击鼠标右键才会弹出相应的快捷菜单（Eclipse 中的文件都是 IFile 的子类）。

（3）在 objectContribution 节点中添加 menu 子节点。

（4）在 objectContribution 节点中添加 action 子节点，设定 action 节点相关属性，如图 16-10 所示。

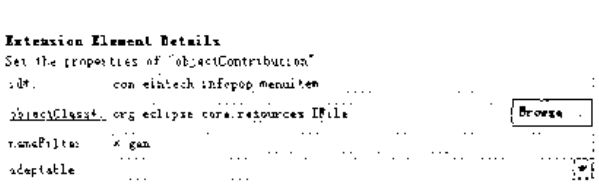


图 16-9　objectContribution 属性设置

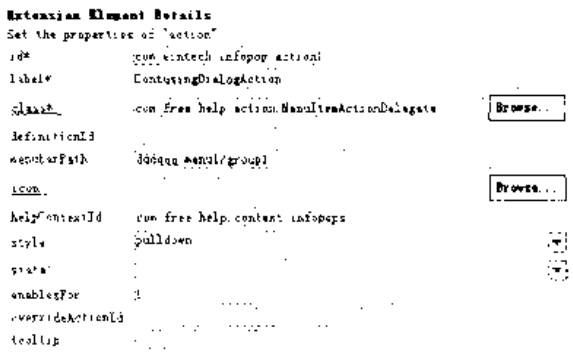


图 16-10　action 节点属性

上下文帮助插件的 plugin.xml 如例程 16-10 所示。

例程 16-10　plugin.xml

```
<plugin>

<extension
       name="Confusing Input Dialog"
       point="org.eclipse.ui.popupMenus">

    <objectContribution
        id="com.eintech.infopop.menuitem"
        nameFilter="*.gan"
        objectClass="org.eclipse.core.resources.IFile">
        <menu
          label="New Submenu"
          path="additions"
          id="dddqqq.menu1">
        <separator
            name="group1">
        </separator>
      </menu>
      <action
          class="com.free.help.action.MenuItemActionDelegate"
          enablesFor="1"
          helpContextId="com.free.help.context.infopops"
          id="com.eintech.infopop.action1"
          label="ConfusingDialogAction"
          menubarPath="dddqqq.menu1/group1"
          style="pulldown"/>
    </objectContribution>
  </extension>
  <extension
      point="org.eclipse.help.contexts">
    <contexts file="contexts_Confusing.xml"/>
  </extension>

</plugin>
```

Action 的实现类为“MenuItemActionDelegate”，MenuItemActionDelegate 继承于“org.eclipse.jface.dialogs.InputDialog”（代表了一个输入的对话框），实现了“IObjectActionDelegate”接口。Action 实现类代码如例程 16-11 所示。

例程 16-11　MenuItemActionDelegate.java

```
package com.free.help.action;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.InputDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Text;
```

```
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.help.WorkbenchHelp;

public class MenuItemActionDelegate extends InputDialog
        implements
            IObjectActionDelegate {

    public MenuItemActionDelegate() {
        super(PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell(),
                "What is the answer to life, the universe and everything?",
                "Enter your answer here:", "gan", null);

    }

    public void setActivePart(IAction action, IWorkbenchPart targetPart) {
    }

    public void run(IAction action) {
        this.open();
    }

    public void selectionChanged(IAction action, ISelection selection) {
    }

    protected Control createDialogArea(Composite parent) {
        Composite dialogArea = (Composite) super.createDialogArea(parent);
        Text inputField = getText();
        WorkbenchHelp.setHelp(inputField, "com.free.help.context.infopops");
        return dialogArea;
    }

}
```

MenuItemActionDelegate 类中通过 WorkbenchHelp.setHelp 设置上下文帮助，参数为“(inputField,"com.free.help.context.infopops")”，其中“inputField”为组件，“com.free.help.context.infopops”为上下文的 ID。

16.3.3 运行上下文帮助插件

此插件运行后，用户只有在扩展名为 gan、对象类型为“org.eclipse.core.resources.IFile”的文件上单击鼠标右键才会弹出相应的快捷菜单，运行步骤如下。

（1）在 PDE 中运行插件。

（2）建立一个项目。

（3）建立一个扩展名为 gan 的文件。

（4）在此文件上单击鼠标右键，弹出的快捷菜单如图 16-11 所示。

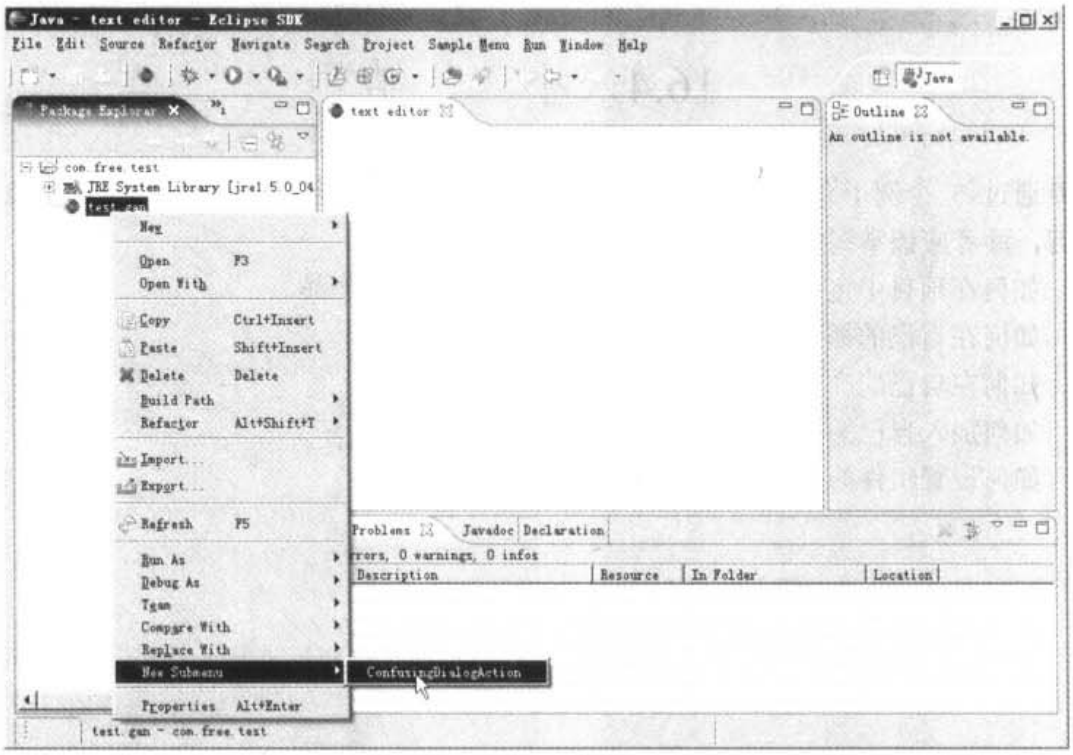


图 16-11　弹出的快捷菜单

（5）单击“ConfusingDialogAction”菜单项，弹出对话框。在输入框中按【F1】键，弹出上下文帮助信息，如图 16-12 所示。

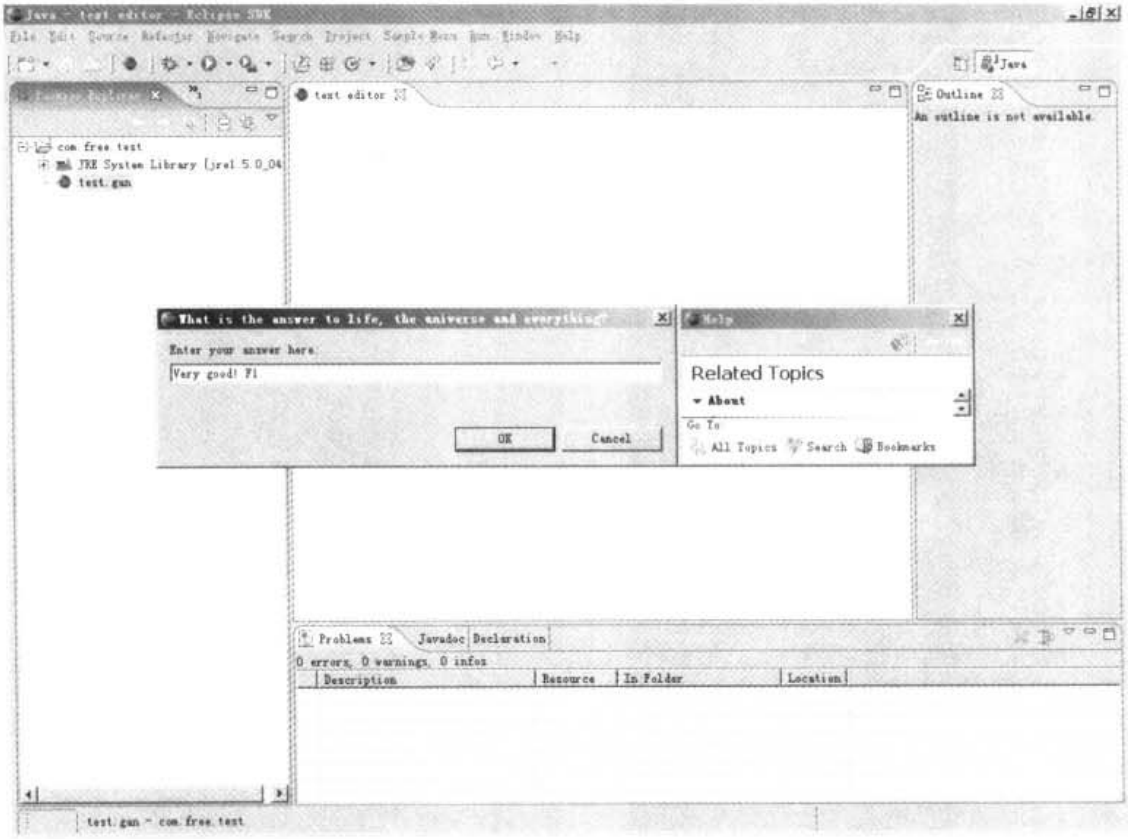


图 16-12　上下文帮助信息

16.4 小　　结

本章通过 3 个例子介绍了联机帮助、欢迎界面帮助和上下文帮助插件的实现。通过对本章的学习，读者应该掌握如下几方面的内容。

- 如何在项目中定义自己的产品扩展点，运行自己的产品。
- 如何在自己的插件中添加联机帮助。
- 如何在自己的产品中绑定欢迎界面扩展点。
- 如何加入自己的弹出菜单。
- 如何设置组件的上下文信息。

第 17 章　其他扩展点的实现

Eclipse 提供了很多的扩展点，在此无法进行一一分类，有些未分类的扩展点将放在本章实现。

本章将从如下几个方面介绍。

- startup 扩展点的实现
- propertyPages 扩展点的实现

17.1　startup 扩展点的实现

在 Eclipse 的开发中，有一条“懒加载法则”，即只有当需要的时候才会加载插件，所以 Eclipse 不会因为插件的不断膨胀而效率下降，但是有些时候用户会要求当 Eclipse 启动时就调用自己的业务逻辑，Eclipse 为此提供了“org.eclipse.ui.startup”扩展点。

17.1.1　“org.eclipse.ui.startup”扩展点的实现

此扩展点用来注册想要在启动时激活的插件，添加“org.eclipse.ui.startup”扩展点的步骤如下。

（1）在清单文件的 Extensions 页添加“org.eclipse.ui.startup”节点。

（2）在“org.eclipse.ui.startup”节点上添加若干个 startup 子节点，设置 startup 节点的 class 属性为“com.free.other.startup.FreeStartup”。

Startup 插件的清单文件如例程 17-1 所示。

例程 17-1　plugin.xml

```
<plugin>
  <extension
       point="org.eclipse.ui.startup">
    <startup class="com.free.other.startup.FreeStartup"/>
  </extension>

</plugin>
```

17.1.2　“org.eclipse.ui.startup”对应的实现类

插件的实现类“FreeStartup”要实现 IStartup 接口。当插件实现了“org.eclipse.ui.startup”扩展点后，Eclipse 会调用实现类的 earlyStartup()方法，或者调用插件类的 earlyStartup()方法，实现类代码如例程 17-2 所示。

例程 17-2 FreeStartup.java

```
package com.free.other.startup;

import org.eclipse.ui.IStartup;

public class FreeStartup implements IStartup{

    public void earlyStartup() {
        System.out.println("very good! start");
    }

}
```



如果要实现 startup 的实现类，必须实现 org.eclipse.ui.IStartup 接口。一旦启动了工作台，就会从独立的线程中调用 earlyStartup() 方法。 如果 startup 元素具有 class 属性，则将把该类实例化并对此类调用 earlyStartup() 方法。如果未指定 startup 扩展点的实现类，则默认用插件类作为实现类，但不需要指定扩展点的 class 值，否则插件类将被实例化两次（一次由常规插件激活操作实例化，一次由 startup 机制实例化）。

17.1.3 “org.eclipse.ui.startup”对应插件运行效果

当工作台启动后，Eclipse 会调用 startup 扩展点的实现类的 earlyStartup()方法，插件运行后会在宿主工作台输出“very good! start”的信息，运行效果如图 17-1 所示。

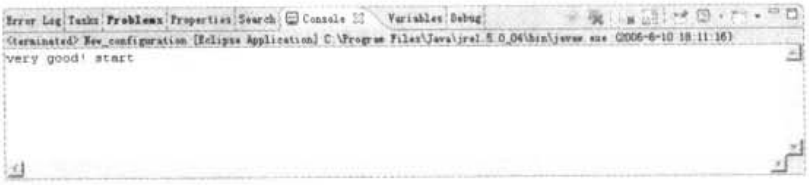


图 17-1 startup 插件输出信息

17.2 “propertyPages”扩展点的实现

在 Eclipse 中，每种资源都有相关的属性。当选定资源后，用户可以打开 properties 视图查看资源的属性，但有些用户自定义的属性，Eclipse 的 properties 视图是不会显示的。Eclipse 提供了“org.eclipse.ui.propertyPages”的扩展点，用户可以定制属性的对话框来显示自己关心的属性。

17.2.1 “org.eclipse.ui.propertyPages”扩展点的实现

此扩展点用来为给定类型的对象添加其他属性页面。一旦定义了这些属性页面，它们就会出现在该类型对象的“属性对话框”中。

属性页面是用户与对象的属性交互的友好方法。与 properties 视图（它限制可用于编辑对象属性的空间）不同，属性页面的有利之处在于具有使用标签、图标等来定义更大、更复杂控件的自由。还可将逻辑上一致的各属性群集在一个页面中，而不是分散在属性表中。在大多数应用程序中，通过属性视图显示对象的某些属性，而通过属性页面显示另一些属性是很恰当的。

添加“org.eclipse.ui.propertyPages”扩展点的步骤如下。

（1）在清单文件的 Extensions 页添加“org.eclipse.ui.propertyPages”节点。

（2）在“org.eclipse.ui.propertyPages”节点上添加若干个 page 子节点，设置 page 节点的属性，如图 17-2 所示。

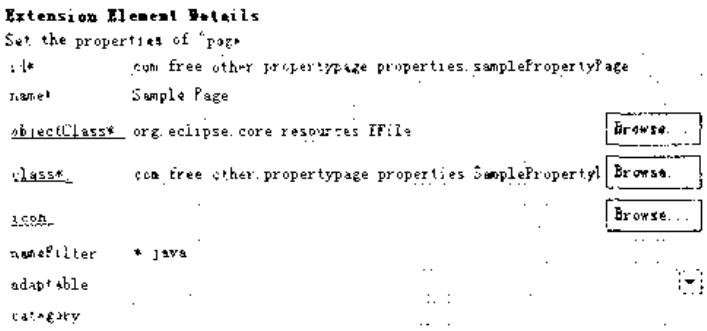


图 17-2　page 节点属性

通过 page 节点的属性可以看出，只有当选定 IFile 类型的对象并且扩展名为 Java 的文件才会显示相关的属性页。插件清单文件如例程 17-3 所示。

例程 17-3　plugin.xml

```
<plugin>

  <extension
      point="org.eclipse.ui.propertyPages">
    <page
        objectClass="org.eclipse.core.resources.IFile"
        name="Sample Page"
        nameFilter="*.java"
        class="com.free.other.propertypage.properties.SamplePropertyPage"
        id="com.free.other.propertypage.properties.samplePropertyPage">
    </page>
  </extension>

</plugin>
```

17.2.2　“org.eclipse.ui.propertyPages”对应的实现类

扩展点对应的实现类为“SamplePropertyPage”。SamplePropertyPage 类继承于 PropertyPage 类，通过 createContents(Composite parent)方法构建属性页。实现类代码如例程 17-4 所示。

例程 17-4 SamplePropertyPage.java

```
package com.free.other.propertypage.properties;

import org.eclipse.core.resources.IResource;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.QualifiedName;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.dialogs.PropertyPage;

public class SamplePropertyPage extends PropertyPage {

    private static final String PATH_TITLE = "Path:";
    private static final String OWNER_TITLE = "&Owner:";
    private static final String OWNER_PROPERTY = "OWNER";
    private static final String DEFAULT_OWNER = "John Doe";

    private static final int TEXT_FIELD_WIDTH = 50;

    private Text ownerText;

    public SamplePropertyPage() {
        super();
    }

    private void addFirstSection(Composite parent) {
        Composite composite = createDefaultComposite(parent);

        Label pathLabel = new Label(composite, SWT.NONE);
        pathLabel.setText(PATH_TITLE);

        Text pathValueText = new Text(composite, SWT.WRAP | SWT.READ_ONLY);
        pathValueText.setText(((IResource) getElement()).getFullPath().toString());
    }

    private void addSeparator(Composite parent) {
        Label separator = new Label(parent, SWT.SEPARATOR | SWT.HORIZONTAL);
        GridData gridData = new GridData();
        gridData.horizontalAlignment = GridData.FILL;
        gridData.grabExcessHorizontalSpace = true;
        separator.setLayoutData(gridData);
    }

    private void addSecondSection(Composite parent) {
        Composite composite = createDefaultComposite(parent);
```

```
        Label ownerLabel = new Label(composite, SWT.NONE);
        ownerLabel.setText(OWNER_TITLE);

        ownerText = new Text(composite, SWT.SINGLE | SWT.BORDER);
        GridData gd = new GridData();
        gd.widthHint = convertWidthInCharsToPixels(TEXT_FIELD_WIDTH);
        ownerText.setLayoutData(gd);

        try {
            String owner =
                ((IResource) getElement()).getPersistentProperty(
                    new QualifiedName("", OWNER_PROPERTY));
            ownerText.setText((owner != null) ? owner : DEFAULT_OWNER);
        } catch (CoreException e) {
            ownerText.setText(DEFAULT_OWNER);
        }
    }
    //构造属性页组件
    protected Control createContents(Composite parent) {
        Composite composite = new Composite(parent, SWT.NONE);
        GridLayout layout = new GridLayout();
        composite.setLayout(layout);
        GridData data = new GridData(GridData.FILL);
        data.grabExcessHorizontalSpace = true;
        composite.setLayoutData(data);

        addFirstSection(composite);
        addSeparator(composite);
        addSecondSection(composite);
        return composite;
    }

    private Composite createDefaultComposite(Composite parent) {
        Composite composite = new Composite(parent, SWT.NULL);
        GridLayout layout = new GridLayout();
        layout.numColumns = 2;
        composite.setLayout(layout);

        GridData data = new GridData();
        data.verticalAlignment = GridData.FILL;
        data.horizontalAlignment = GridData.FILL;
        composite.setLayoutData(data);

        return composite;
    }

    //还原默认值
    protected void performDefaults() {
        ownerText.setText(DEFAULT_OWNER);
    }
```

```
        //持久化用户输入的信息
        public boolean performOk() {
            try {
                ((IResource) getElement()).setPersistentProperty(
                    new QualifiedName("", OWNER_PROPERTY),
                    ownerText.getText());
            } catch (CoreException e) {
                return false;
            }
            return true;
        }

    }
```

17.2.3 “org.eclipse.ui.propertyPages”对应插件运行效果

此插件运行后，用户只有对扩展名为 java、对象类型为“org.eclipse.core.resources.IFile”的文件才会显示相关属性，运行步骤如下。

（1）在 PDE 中运行插件。

（2）建立一个项目。

（3）建立一个扩展名为 java 的文件。

（4）在此文件上单击鼠标右键菜单，选择“Properties”菜单，打开的窗口如图 17-3 所示。

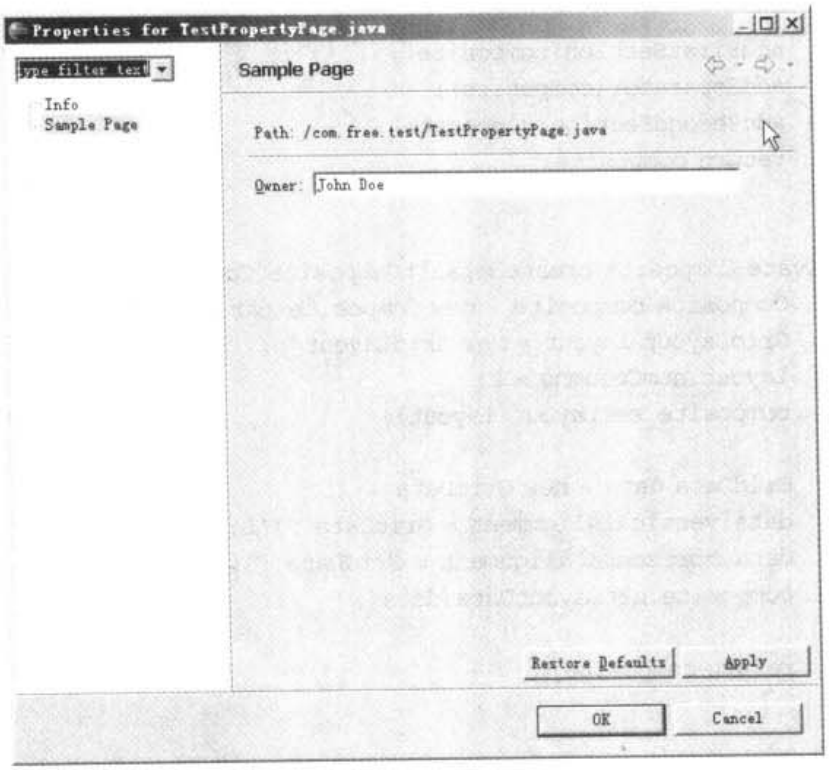


图 17-3　属性对话框

17.3　小　　结

Eclipse 是建立在插件之上的一个大拼图，它提供的扩展点非常多，打开 Eclipse 的帮助文档，在平台扩展点页帮助中列出了一百多个扩展点。本书只提供了常用的一些扩展点的介绍，对插件开发有兴趣的读者可以查找 Eclipse 的帮助文档，从帮助中可以得到更多的信息。

通过学习本章，读者应该掌握如下几方面的内容。

- 如何在项目中添加 startup 扩展点，以及添加的时机。
- 如何在项目中添加 propertyPages 扩展点。

第 18 章　插件开发辅助功能

在 Eclipse 插件的开发过程中，还需要掌握一些插件开发的辅助功能，虽然不需要用户直接实现插件扩展点，但在插件的开发中是经常用到的。

本章将介绍如下几个方面的内容。

- 属性视图
- 大纲视图
- 构造 EditorInput
- 操纵编辑器
- 插件国际化

18.1　属 性 视 图

Eclipse 中提供了标准的属性视图窗口，方便用户对当前选择的对象属性进行编辑，用户可以通过“Window”→“Show view”→“Other”菜单，在“Show View”窗口中选择属性视图项，如图 18-1 所示。



图 18-1　选择属性视图

属性视图提供了对所选节点属性进行编辑的功能。例如，用户在自定义的资源窗口中定义了项目、目录和叶子节点。当选定节点后，如果想通过 Eclipse 提供的视图编辑所选对象的属性，就要实现 Eclipse 的属性视图。

18.1.1 IPropertySource 接口

要实现 Eclipse 视图，用户要让所选节点对象实现 IPropertySource 接口。通过适配 IPropertySource 接口，Eclipse 能帮用户创建属性视图。IPropertySource 接口定义如例程 18-1 所示。

例程 18-1 IPropertySource.java

```
public interface IPropertySource {

    public Object getEditableValue();

    public IPropertyDescriptor[] getPropertyDescriptors();

    public Object getPropertyValue(Object id);

    public boolean isPropertySet(Object id);

    public void resetPropertyValue(Object id);

    public void setPropertyValue(Object id, Object value);
}
```

IPropertySource 接口定义了几个方法，主要实现的方法描述如下。

（1）getPropertyDescriptors：返回 IPropertyDescriptor 数组，数组中每一项表示属性编辑器中的可编辑项，如下代码所示。

```
public IPropertyDescriptor[] getPropertyDescriptors() {
    //定义属性的编辑项
    IPropertyDescriptor[] descriptors = new IPropertyDescriptor[] {
        new TextPropertyDescriptor(PROP_NAME, "Name"),
        new ComboBoxPropertyDescriptor(PROP_VISIBLE, "Visible",
            new String[] { "true", "false" })};
    return descriptors;
}
```

上面代码表示当前选择对象有两个可编辑属性，一个为 TextPropertyDescriptor（文本），另一个为 ComboBoxPropertyDescriptor（下拉列表框）。属性一般有两个参数的构造函数，第一个参数表示当前属性的 ID 值（唯一的标识），另一个参数为属性的显示名称。

（2）setPropertyValue：表示根据参数的 ID 给当前对象赋值。当用户通过属性视图编辑对象的属性后，将会调用 setPropertyValue 给所选对象赋值，如下代码所示。

```
public void setPropertyValue(Object id, Object value) {
//设置属性的值
    if (PROP_NAME.equals(id))
        setName((String) value);
    if (PROP_VISIBLE.equals(id))
        setVisible(((Integer) value).intValue() == 0);
}
```

setPropertyValue 方法的 id 值为在 IPropertyDescriptor 数组对象中定义的标识，value 值为修改后的属性值，上面代码表示通过 id 给对象对应的属性赋值。

（3）getPropertyValue：表示根据 id 值获得对象的某个属性的值，Eclipse 在显示属性视图时，将遍历所有的属性，获得每个属性的显示值，如下代码所示。

```
public Object getPropertyValue(Object id) {
    //得到属性编辑项的值
   if (PROP_NAME.equals(id))
      return getName();
   if (PROP_VISIBLE.equals(id))
      return isVisible() ? new Integer(0) : new Integer(1);
   return null;
}
```

18.1.2　属性视图的实现

定义属性视图的显示非常简单，用户只要实现 IPropertySource 接口方法即可，如例程 18-2 所示。

例程 18-2　Node.java

```
package com.example.model;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.draw2d.geometry.Point;
import org.eclipse.ui.views.properties.ComboBoxPropertyDescriptor;
import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.IPropertySource;
import org.eclipse.ui.views.properties.PropertyDescriptor;
import org.eclipse.ui.views.properties.TextPropertyDescriptor;

public class Node extends Element implements IPropertySource {

    //定义属性的常量ID
   final public static String PROP_NAME = "NAME";
   final public static String PROP_VISIBLE = "VISIBLE";

   protected String name = "Node";

   protected boolean visible = true;


   public boolean isVisible() {
      return visible;
   }

   public void setVisible(boolean visible) {
      if (this.visible == visible) {
```

```
            return;
        }
        this.visible = visible;
        firePropertyChange(PROP_VISIBLE, null, Boolean.valueOf(visible));
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (this.name.equals(name)) {
            return;
        }
        this.name = name;
        //当模型名字改变后，通知监听器更新名字的显示
        firePropertyChange(PROP_NAME, null, name);
    }

    //得到可编辑的值
    public Object getEditableValue() {
        return this;
    }

    //得到属性编辑项
    public IPropertyDescriptor[] getPropertyDescriptors() {

        List propertyList = new ArrayList();
        propertyList.add(new TextPropertyDescriptor(PROP_NAME, "Name"));
        propertyList.add(new ComboBoxPropertyDescriptor(PROP_VISIBLE, "Visible",
                new String[] { "true", "false" }));

        //定义分组
        String groupString = "group";
        PropertyDescriptor pd1 = new TextPropertyDescriptor("grouptest1", "test1");
        pd1.setCategory(groupString);
        propertyList.add(pd1);

        PropertyDescriptor pd2 = new TextPropertyDescriptor("grouptest2", "test2");
        pd2.setCategory(groupString);
        propertyList.add(pd2);

        return (IPropertyDescriptor[]) propertyList.toArray
                (new IPropertyDescriptor[propertyList.size()]);
    }

    public Object getPropertyValue(Object id) {
      //得到属性编辑项的值
        if (PROP_NAME.equals(id))
            return getName();
        if (PROP_VISIBLE.equals(id))
```

```
            return isVisible() ? new Integer(0) : new Integer(1);
        return null;
    }

    public boolean isPropertySet(Object id) {
        return true;
    }

    public void resetPropertyValue(Object id) {

    }

    public void setPropertyValue(Object id, Object value) {
      //设置属性的值
        if (PROP_NAME.equals(id))
            setName((String) value);
        if (PROP_VISIBLE.equals(id))
            setVisible(((Integer) value).intValue() == 0);
    }
}
```

Node 节点实现了 IPropertySource 接口，当用户选择了 Node 对象时，属性视图中将会显示 Node 节点对应的属性，用户能修改对应的属性。另外，Node 节点还通过 PropertyDescriptor 的 setCategory 方法设置了属性的分组，通过这种方式，可以把同一属性页中相关属性放入同一分组中显示。Node 的属性页显示如图 18-2 所示。

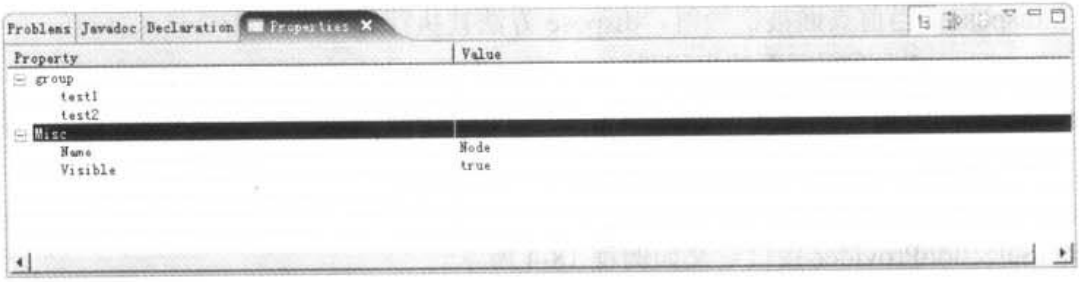


图 18-2　属性页的显示

18.2　大 纲 视 图

大纲视图也是 Eclipse 提供的标准视图。大纲视图一般通过结构化的图形表现对象的结构，例如 Java 文件编辑器实现的大纲视图。用户可以通过大纲视图查看 Java 类的方法、属性和类的结构，另外还可以通过选择相应节点定位到 Java 类的相应位置。用户可以通过“Show View”窗口的 Outline 选择大纲视图。

18.2.1　IContentOutlinePage 接口

用户可以通过适配 IContentOutlinePage 接口实现大纲视图。IContentOutlinePage 接口继承于 IPage 和 ISelectionProvider 接口，没有定义自己的方法。

用户不应该直接实现 IPage 接口，IPageBookViewPage 继承了 IPage 接口实现页面视图。IPage 接口的定义如例程 18-3 所示。

例程 18-3 IPage.java

```
package org.eclipse.ui.part;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.ui.IActionBars;

public interface IPage {

   public void createControl(Composite parent);

   public void dispose();

   public Control getControl();

   public void setActionBars(IActionBars actionBars);

   public void setFocus();
}
```

IPage 定义了 5 个方法，方法的描述如下。

- createControl：此方法创建当前大纲页面中指定的父组件，即当前大纲的根组件。
- dispose：当前页面被销毁时，dispose 方法被执行。
- getControl：返回页面的根组件。
- setActionBars：允许页面创建特定的“Action Bar”（菜单和工具栏等）。
- setFocus：当页面获得焦点时执行。

另外 IContentOutlinePage 接口还继承于 ISelectionProvider 接口，提供了用户选择对象的监听能力。ISelectionProvider 接口定义如例程 18-4 所示。

例程 18-4 ISelectionProvider.java

```
package org.eclipse.jface.viewers;

public interface ISelectionProvider {
   public void addSelectionChangedListener(ISelectionChangedListener listener);

   public ISelection getSelection();

   public void removeSelectionChangedListener(
         ISelectionChangedListener listener);

   public void setSelection(ISelection selection);
}
```

当对象实现了 ISelectionProvider 接口后，用户能够为对象添加选择事件的监听器和设置选择项等。ISelectionProvider 定义方法描述如下。

- addSelectionChangedListener：添加选择对象改变的监听器。
- getSelection：返回选择项。
- removeSelectionChangedListener：删除相应的选择事件的监听器。
- setSelection：设置选择项。

18.2.2　大纲视图的实现

Eclipse 的大纲视图不是通过实现相应的扩展点实现的，而是通过 Adapter（适配器）动态适配的。Eclipse 的大多数核心类都实现 IAdaptable 接口，能通过 IAdaptable 接口的 getAdapter 方法动态适配接口不相容的对象。

在 Eclipse 的编辑器中可以通过 getAdapter 方法得到编辑器相关联的大纲视图，如以下代码所示。

```
public Object getAdapter(Class type) {
   if (type == IContentOutlinePage.class)
      return new OutlinePage();
   return super.getAdapter(type);
}
```

其中，如果要适配的是 IContentOutlinePage 的对象，编辑器会直接创建 OutlinePage 类型的大纲视图。

虽然实现 IContentOutlinePage 接口的对象表示大纲视图，但用户不应该直接实现 IContentOutlinePage 接口，可以通过 IContentOutlinePage 的子类完成大纲视图的主要功能，在此基础上再实现自己相应的大纲视图。

GEF 中的 ContentOutlinePage 类实现了 IContentOutlinePage 接口，用户可以通过继承 ContentOutlinePage 类实现 GEF 相关的大纲视图，如例程 18-5 所示。

例程 18-5　OutlinePage.java

```
class OutlinePage extends ContentOutlinePage {

   private Control outline;

   public OutlinePage() {
    //创建大纲视图的 EditPart Viewer
      super(new TreeViewer());
   }

   public void init(IPageSite pageSite) {
      super.init(pageSite);
   }

   public void createControl(Composite parent) {
    //创建大纲
      outline = getViewer().createControl(parent);
      getSelectionSynchronizer().addViewer(getViewer());
      getViewer().setEditDomain(getEditDomain());
```

```
            //添加大纲 EditPart Factory
            getViewer().setEditPartFactory(new TreePartFactory());
            //设置大纲的内容
            getViewer().setContents(getDiagram());
        }

        public Control getControl() {
            return outline;
        }

        public void dispose() {
            getSelectionSynchronizer().removeViewer(getViewer());
            super.dispose();
        }
    }
```

以上例程中只是简单地创建了 GEF 相关大纲父组件。另外，在不同的编辑器（非 GEF 编辑器）中，用户可以通过相关的大纲实现类实现自己的功能，上例大纲视图运行效果如图 18-3 所示。

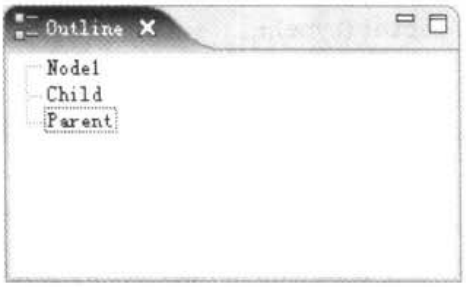


图 18-3　大纲视图

18.3　构造 EditorInput

在前面介绍编辑器的章节中，为了让读者更好地了解编辑器的实现，编辑器都是以文件作为编辑器的输入，让用户通过文件浏览器打开文件作为输入。试想一下，如果用户要实现某个 XML 文件的某个节点的内容作为编辑器的输入，同一个文件由多个编辑器进行表现，或者多个不同的文件要作为同一个编辑器的输入（例如，一个图形保存在多个文件中）。类似这种需求是非常多的，也是无法预测的，如果要实现类似的功能，就要用到 Eclipse 提供的 IEditorInput 了。

通过 IEditorInput 对象，用户可以自由组织自己的输入，从而把编辑器和所依赖的输入完全分离出来，用户只需要输入转换成编辑器所依赖的 IEditorInput 对象就可以了。

18.3.1　IEditorInput 接口

IEditorInput 对象代表了打开编辑器的输入，例如前面介绍的编辑器通过磁盘文件（IFileEditorInput）作为输入，在编辑器初始化的时候可以从文件中读取编辑器的内容。

要想自定义编辑器的输入，用户要实现 IEditorInput 接口。通过 IEditorInput 对象，编辑器

能够初始化编辑器的显示内容。另外，每个 IEditorInput 对象是唯一的，可以通过 IEditorInput 对象区分每一个编辑器打开的不同实例。IEditorInput 接口定义如例程 18-6 所示。

例程 18-6　IEditorInput.java

```
package org.eclipse.ui;

import org.eclipse.core.runtime.IAdaptable;
import org.eclipse.jface.resource.ImageDescriptor;

public interface IEditorInput extends IAdaptable {
   public boolean exists();

   public ImageDescriptor getImageDescriptor();

   public String getName();

   public IPersistableElement getPersistable();

   public String getToolTipText();
}
```

IEditorInput 接口继承于 IAdaptable，另外 IEditorInput 定义了几个方法，描述如下。

- exists：返回编辑器的输入是否存在，主要用于是否把文件显示在最近使用的文件菜单中。
- getImageDescriptor：返回编辑器的 ImageDescriptor 对象。
- getName：返回编辑器的名字。
- getPersistable：返回编辑器被持久化的对象。
- getToolTipText：返回编辑器的提示信息。

18.3.2　IEditorInput 的实现

IEditorInput 接口只定义了 Eclipse 能从编辑器输入读出的基本信息，要想实现用户的输入，需要扩展 IEditorInput 接口，添加用户的输入信息，以便在编辑器初始化的时候读出用户自定义的输入，从而初始化编辑器。实现代码如例程 18-7 所示。

例程 18-7　KenoahEditorInput.java

```
package com.kenoah.kde.core.input;

import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IPersistableElement;

import com.kenoah.kde.core.KenoahProject;

public class KenoahEditorInput implements IEditorInput {

    //当前编辑器的名称
```

```
private String Name;

//初始化当前编辑器的数据
private Object data;

//当前编辑器所属的项目
private KenoahProject kenoahProject;

public Object getData() {
    return data;
}

public void setData(Object data) {
    this.data = data;
}

public KenoahProject getKenoahProject() {
    return kenoahProject;
}

public void setKenoahProject(KenoahProject kenoahProject) {
    this.kenoahProject = kenoahProject;
}

public String getName() {
    return Name;
}

public void setName(String name) {
    Name = name;
}

public boolean exists() {
    return false;
}

public ImageDescriptor getImageDescriptor() {
    return null;
}

public IPersistableElement getPersistable() {
    return null;
}

public String getToolTipText() {
    return getName();
}

public Object getAdapter(Class adapter) {
    return null;
}
```

```
}
```

上例实现了 IEditorInput 接口，实现非常简单，只加入了当前编辑器所属的项目和编辑器初始化的对象，但这样已经足够了。用户可以按自己的需求，把数据转换成相应的输入（这些数据可以是磁盘文件，也可以是其他的对象），编辑器实例可以通过读取 EditorInput 中的数据初始化编辑器。

18.4　操纵编辑器

当用户创建了自己的编辑器输入后，接下来要做的事情是如何通过自己的输入打开编辑器、如何判断和当前输入相关联的编辑器（一个输入对应一个编辑器）是否打开，以及如何在程序中关闭相应的编辑器。

IWorkbenchPage 接口提供了查找、打开和关闭编辑器的相关方法，编辑器（EditorPart）和视图（ViewPart）都可以通过 IWorkbenchSite 的 getPage 方法得到当前工作台的 IWorkbenchPage 对象。下面将介绍如何在程序中查找、打开和关闭编辑器窗口。

18.4.1　查找编辑器

用户可以通过 IWorkbenchPage 接口定义的 findEditor 方法查找和 IEditorInput 相关联的编辑器。IWorkbenchPage 有一个实现类 WorkbenchPage，WorkbenchPage 通过 EditorManager 来实现编辑器的查找，代码如下所示。

```
public IEditorPart findEditor(IEditorInput input) {
    return getEditorManager().findEditor(input);
}
```

findEditor 方法输入的参数为 EditorInput 对象，返回的是与编辑器相关联的 EditorPart 对象（即编辑器实例），有兴趣的读者可以通过 EditorManager 研究于 Eclipse 中查找编辑器的实现。

通过 EditorInput 对象，Eclipse 能找到相关的 EditorPart，但如果两个不同的 EditorInput 对象都是对应的同一个编辑器（虽然 EditorInput 对象不同，但 EditorInput 对象中包含的输入是一样的，例如都指向同一个文件），用户应该比较 EditorInput 中的输入对象，从而确定是新建还是打开上一次的编辑器，如下片段代码所示。

```
public static IEditorPart getPart(KenoahEditorInput input, IWorkbenchPage page) {
    //获得 EditorInput 的输入对象
    Object data = input.getData();
    //获得 EditorInput 中对象的项目
    KenoahProject kp = input.getKenoahProject();
    IEditorPart part = null;
    //得到当前 IWorkbenchPage 中所有的编辑器的引用
    IEditorReference[] references = page.getEditorReferences();
    //遍历所有的编辑器，查找和输入相关的编辑器是否存在
    for (int n = 0; n < references.length; n++) {
    //得到编辑器
```

```
                IEditorPart editor = getEditor(references[n]);
                if (editor != null) {
                   //得到编辑器关联的 EditorInput
                   IEditorInput editorInput = editor.getEditorInput();
                   //判断 EditorInput 是否为关心的 EditorInput
                   if (editorInput instanceof KenoahEditorInput) {
                      KenoahEditorInput input1 = (KenoahEditorInput) editorInput;
                      //判断输入的数据是否相同
                      if ((input1.getData() == data) && input1.getKenoahProject() == kp) {
                       //如果输入相同，则表明两个 EditorInput 对应同一个编辑器
                         part = editor;
                         break;
                      }
                   }
                }
             }
             //返回编辑器实例
             return part;
          }
```

上面代码演示了如何从 WorkbenchPage 中查找某个 EditorInput 是否已经打开，如果打开了，则返回编辑器对应的实例，从而实现了某个实例的编辑器是否打开的功能。

18.4.2 打开和关闭编辑器

当用户查找和 EditorInput 对应的编辑器存在的时候，应该把编辑器设置为当前编辑器并放在其他编辑器之前，如果编辑器不存在时，要重新打开编辑器。另外，如果和编辑器相关的数据被销毁时（例如文件被删除）要关闭相关的编辑器，IWorkbenchPage 中相关的方法如下。

（1）bringToTop：定义为“public void bringToTop(IWorkbenchPart part)”，表示当用户查找和 EditorInput 对应的编辑器存在的时候，通过 bringToTop 方法能把编辑器放在其他编辑器之前，并设置为当前编辑器。

（2）openEditor：定义为“public IEditorPart openEditor(IEditorInput input, String editorId)”，表示用 input 作为输入，打开 editorId 对应的编辑器。editorId 为编辑器实现扩展点的 ID 号。

（3）closeEditor：定义为“public boolean closeEditor(IEditorPart editor, boolean save)”，表示关闭 editor 所代表的编辑器。另外，save 表示是否保存编辑器的内容。



IWorkbenchPage 定义了工作台编辑器和视图的相关操作，也可以通过 findView 方法找到指定视图的引用。

18.5 插件国际化

国际化 I18N 是英文单词“internationalization”的 i 和 n 之间的 18 个字母的缩写。Eclipse 是功能非常完善的软件，用户可以对开发的插件提供国际化的支持。Eclipse 中插件的国际化可以通过一组单独的插件完成，例如，Eclipse 提供的中文汉化包就是一组插件（Fragment，

插件的另一种形式）。

插件的国际化包括实现类的国际化和插件清单文件的国际化，它们的实现方式不完全一样，后面将会分别介绍。

18.5.1　国际化插件的创建

在 Eclipse 中，一切辅助功能都是通过插件的形式提供的，语言包也是一种插件。用户可以通过创建语言包插件，并通过属性文件提供插件国际化能力。给插件提供国际化的步骤如下。

（1）创建被国际化的插件（前面介绍的插件都可以被国际化）。

（2）创建语言包插件：通过“New”菜单新建“Fragment Project”创建语言包插件，并输入新建项目的名字，最后通过“Browse”按钮选择语言包插件所依附的插件（即要对哪个插件国际化），如图 18-4 所示。

（3）提取要被国际化的文字，放入 properties 文件中：语言包可以提供多国语言的支持，用户在运行时指定所要运行的语言的参数就可以通过此语言展示插件，例如中文的运行参数为“zh”或“zh_CN”。

另外，用户如果要提供多国语言的国际化，就要为所有的语言提供相应的 properties 文件。中文 properties 文件的格式为“名称_标识. properties”，例如“messages_zh.properties”。中文标识的查找顺序为“zh，zh_cn”，本节只针对中文提供国际化的支持，与提供其他语言的支持方法相同，只要把 properties 文件拷贝到目录即可，例如 Eclipse 的汉化包的一个目录下就包含了多种语言的 properties 文件，如图 18-5 所示。

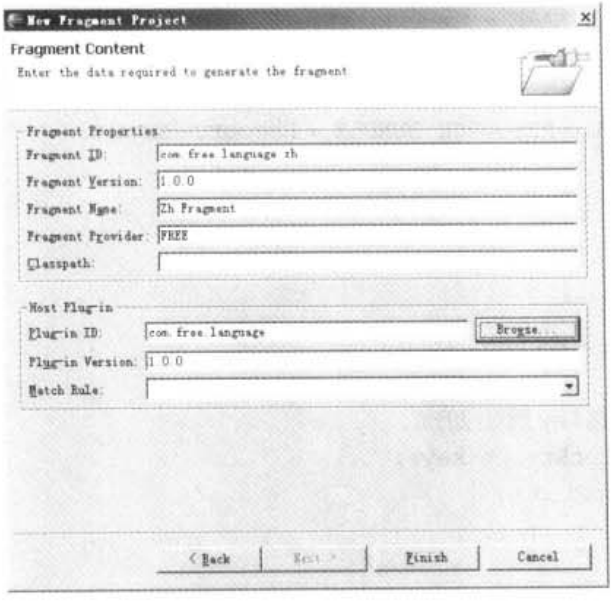


图 18-4　新建语言包插件

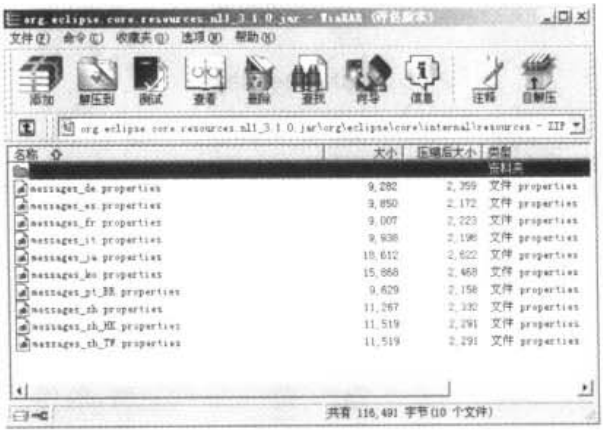


图 18-5　多语言 properties 文件

18.5.2　类的国际化

用户可以为插件的实现类提供国际化的支持，例如菜单的显示可以通过英文和中文两种方式显示。

要实现类的国际化功能，先要把类中要国际化的字符串提取到 properties 文件中，生成键和值的匹配。另外，在实现类中通过 ResourceBundle 得到 properties 文件中键所对应的值。类的国际化步骤如下。

（1）根据语言生成国际化版本的 properties 文件。

properties 文件中包含键和值的属性对，如果生成中文的字符串，要通过“native2ascii.exe”把中文转换成 Unicode 编码。如下为英文和中文的 properties 文件的国际化片段信息。

- 英文 messages.properties 文件

```
com.free.name=I18N
com.free.message=Hello, Plugin
```

- 中文 messages_zh.properties 文件

```
com.free.name=\u56FD\u9645\u5316\u63D2\u4EF6
com.free.message=Hello, \u56FD\u9645\u5316
```

（2）封装 ResourceBundle 类，实现从 properties 文件获取数据的能力，如例程 18-8 所示。

例程 18-8 Messages.java

```
package com.free.language.actions;

import java.util.MissingResourceException;
import java.util.ResourceBundle;

public class Messages {
    //properties 文件的名字
    private static final String BUNDLE_NAME = "messages";

    //得到RESOURCE_BUNDLE
    private static final ResourceBundle RESOURCE_BUNDLE = ResourceBundle
            .getBundle(BUNDLE_NAME);

    private Messages() {
    }

    public static String getString(String key) {
        try {
            //通过RESOURCE_BUNDLE 得到Key 所对应的值
            return RESOURCE_BUNDLE.getString(key);
        } catch (MissingResourceException e) {
            return '!' + key + '!';
        }
    }
}
```

用户通过 getString 方法能够获得 properties 文件中的对应键值。

（3）在插件实现类中调用国际化的字符串：通过上例的 Messages.getString ("com.free.name")方法可以得到相应的对应值。

```
public void run(IAction action) {
    //通过Messages 得到com.free.name 所对应的信息
```

```
        MessageDialog.openInformation(
            window.getShell(),
            Messages.getString("com.free.name"),
            Messages.getString("com.free.message"));
    }
```

在 Eclipse 中提供了一种比较快速的方式国际化，用户可以在现有的实现类中提取字符串，可以通过鼠标右键菜单“Source”→“Externalize Strings”打开对话框，提取当前类中的字符串进行国际化。在打开的对话框中有要国际化的键值，输入对应的值、properties 文件和对应的 Message 文件，如图 18-6 所示。



图 18-6　提取字符串

另外，通过此方式，Eclipse 会把中文字符串相应的 Unicode 编码。

18.5.3　plugin.xml 文件格式化

plugin.xml 文件国际化的 properties 文件格式和类的 properties 文件相同，但文件名和引用的方式有差别。

（1）plugin.xml 文件对应的 properties 文件为 plugin.properties 和 plugin_zh.properties，且放在与 plugin.xml 文件对应的目录中。

（2）引用 properties 文件的方式不同，在 plugin.xml 文件中只需要通过“%”符号就可以引用相应 properties 文件中的值，例如“label="%com.free.plugin.lable"”可以引用 properties 文件中的“com.free.plugin.lable”关键字对应的值。

18.5.4　调试国际化插件

如果要显示国际化插件的运行效果，可以设置调试参数即可，例如 properties 文件标识为“zh”，可以通过设置调试的“Program Arguments”参数为“-nl zh”，运行中文插件，如图 18-7 所示。

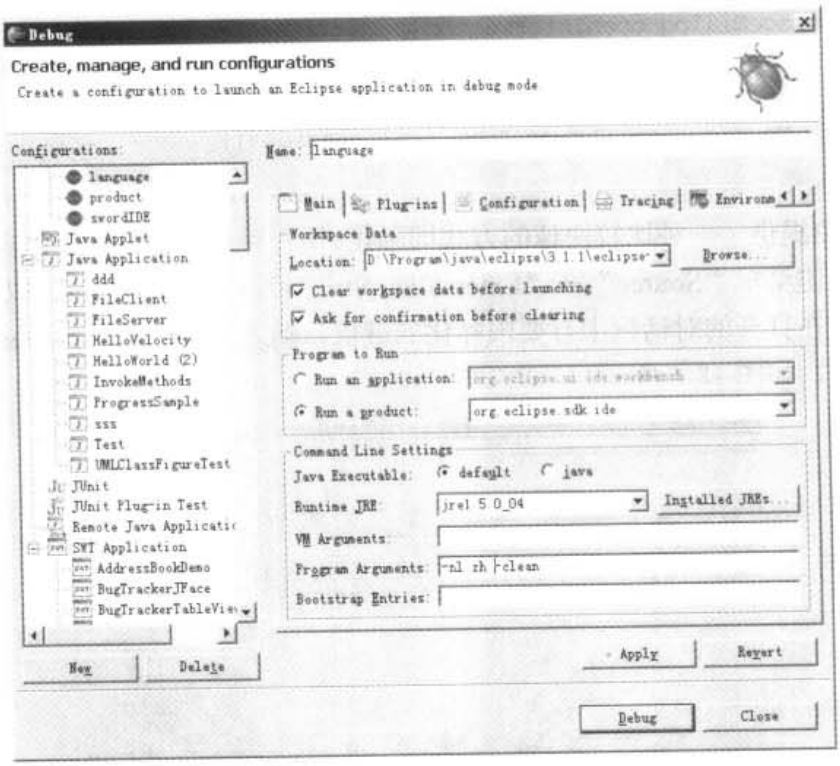


图 18-7　调试中文插件

如果要运行为英文插件，只需要把参数“Program Arguments”设置为“-nl en”即可，读者可以在本书附带光盘中找到相应的国际化插件的源代码。

18.6　小　　结

本节介绍了属性视图、大纲视图、EditorInput、操纵编辑器和插件国际化的知识，完成这些功能虽然不需要直接实现插件的扩展点，但在插件的开发中是不可缺少的，而且也是插件开发中比较灵活的部分。本章无法介绍全面插件开发的辅助功能，希望读者能在项目的实践中不断摸索。

Eclipse 插件开发方法与实战

第 4 部分　Eclipse 提高篇

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台，就其本身而言，它只是一个框架和一组服务，用于通过插件构建开发环境。Eclipse 是一个非常优秀的平台，在此平台的基础上有非常多的开源框架，通过这些开源的框架，结合插件的开发，用户能构建强大的基于 Eclipse 的应用。

本部分将介绍 GEF（Graphical Editor Framework）、EMF（Eclipse Modelling Framework）、RCP（Rich Client Platform），以及 Eclipse 工作台实现的一些技术，通过这一部分的学习，相信读者会对 Eclipse 插件开发信心大增。

第 19 章　Draw2d 简介及实现

Draw2d 提供了在 SWT 的 Canvas 上展现和布局的能力，GEF 通过 Draw2d 实现 GEF 的视图（GEF 的视图和插件的视图是完全不同的概念）。

本章将从如下几个方面介绍 Draw2d 相关技术的实现。

- Draw2d 简介
- 图形（Figure）
- 连线（Connection）
- UML 关系图实现

19.1　Draw2d 简介

Draw2d 是基于 SWT 的轻量级组件系统，它能提供 GEF 视图层绘制的能力。关于 Draw2d 的简单介绍如下。

19.1.1　Draw2d 简介

Draw2d 是基于 SWT 的轻量级组件系统。Draw2d 的实例由 SWT 的组件、LightweightSystem 和 Draw2d 的 figures 组成。

SWT 的组件是 Draw2d 的容器，figures 是 Draw2d 中的图形，LightweightSystem 是 SWT 和 Draw2d 的桥梁，如图 19-1 所示。

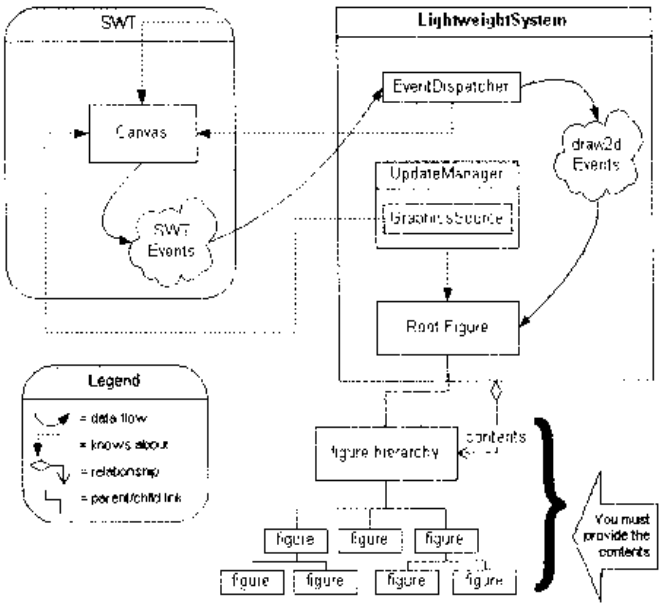


图 19-1　LightweightSystem

IFigure 可以是图形，也可以是容器。IFigure 对象中能加入其他的 IFigure 对象，通过 IFigure 的组合生成 Draw2d 的图形。创建 Draw2d 的程序步骤如下。

（1）创建 SWT 的 Canvas 组件。

（2）添加 LightweightSystem 实例。

（3）添加 IFigure 实例。

（4）把 IFigure 实例加入到 LightweightSystem 中。

19.1.2 “Hello World”实现

一个简单的 Draw2d 实例可以由 Canvas、LightweightSystem 和 IFigure 组成，如例程 19-1 所示。

例程 19-1 HelloWorld.java

```
package com.test.demo;

import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.SWT;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;

public class HelloWorld {
    public static void main(String args[]){
        //新建Shell，Shell是Canvas的子类
        Shell shell = new Shell();
        shell.open();
        shell.setText("Draw2d Hello World");
        //添加LightweightSystem实例
        LightweightSystem lws = new LightweightSystem(shell);
        //添加IFigure实例
        IFigure label = new Label("Hello World");
        //把IFigure添加到LightweightSystem中
        lws.setContents(label);
        Display display = Display.getDefault();
        while (!shell.isDisposed ()) {
            if (!display.readAndDispatch ())
                display.sleep ();
        }
    }
}
```

上例中添加了一个“Hello World”的 Label 标签，程序运行后效果好像和 SWT 的程序没有区别，但本程序已经加入了 Draw2d 的实现，程序运行效果如图 19-2 所示。

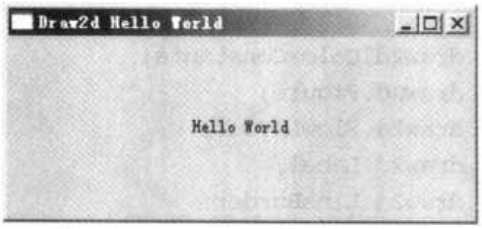


图 19-2　“Hello World”程序

19.2　图　　形

图形(Figure)是 Draw2d 比较重要的部分，它提供了 Draw2d 中视图展现的实现，具体介绍如下。

19.2.1　图形（Figure）的功能

在 Draw2d 中，IFigure 对象代表图形，IFigure 对象还能包含其他的 IFigure 对象，所有的 IFigure 对象组合成用户定制的图形，如图 19-3 所示。

图 19-3 为一个组合的图形。图中 Table 为一个底层的 Figure，在 Table 的 Figure 上面加入了属性和方法的 Figure，通过布局排列 Figure 的位置，如图 19-4 所示。

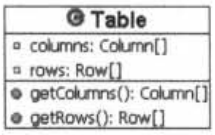


图 19-3　组合图形

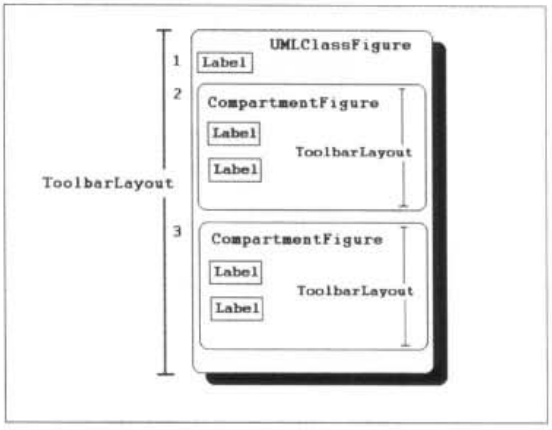


图 19-4　Figure 的组成

Table 的 Figure 为 UMLClassFigure，属性和方法的 Figure 为 CompartmentFigure，通过 ToolbarLayout 垂直排列 Figure 中的元素，Figure 可以通过 setLayoutManager 方法设置当前图形中采用的布局方式。

19.2.2　图形的实现

通常用户可以继承 Figure 类实现自己的图形。Figure 实现了 IFigure 接口，如例程 19-2 所示。

例程 19-2　KCGCommonFigure.java

```
package com.free.table.editor.figure;
```

```
import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.FlowLayout;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.LineBorder;
import org.eclipse.draw2d.PositionConstants;
import org.eclipse.swt.graphics.Image;

public class KCGCommonFigure extends Figure {

    //在Figure中添加标签（标签也是Figure）
    protected Label label = new Label();

    public KCGCommonFigure(String fieldLabel, Image typeIcon) {
        //设置背景色
        setBackgroundColor(ColorConstants.tooltipBackground);
        //设置前景色
        setForegroundColor(ColorConstants.tooltipForeground);
        //设置Figure的边框
        setBorder(new LineBorder());

        //设置label中文字的对齐方式
        label.setTextAlignment(PositionConstants.LEFT);
        //设置label的图标
        label.setIcon(typeIcon);

        //设置Figure的布局方式
        setLayoutManager(new FlowLayout());
        //把label添加到Figure中
        add(label);
        //设置label的显示文字
        setLabelText(fieldLabel);
    }

    public void setLabelText(String newLabel){
        label.setText(newLabel);
    }
}
```

上例中实现了一个 Figure，设置边框为 LineBorder（边框类似为连线），另外还添加了一个 Label 作为显示的 Figure，并设置为 Label 的显示文字和文字对齐方式，以及 Figure 的布局。

19.3 连　　线

连线（Connection）是图形和图形之间的关系，Draw2d 提供了一些连线的实现方式，具体介绍如下。

19.3.1　连线（Connection）的功能

在图形的显示中，连线是比较常用的部分，它表现了图形和图形之间的关系，图形可以通过连线找到与其相关的图形。

图形化应用程序的一个常见任务就是在两个图形之间建立连线。想象一下 UML 类图中的各种连接线，或者程序流程图中表示数据流的线条，它们有着不同的外观，有些连接线还要显示名称，而且最好能不交叉。

利用 Draw2d 中的 Router、Anchor 和 Locator，可以实现多种连接样式。其中 Router 负责连接线的外观和操作方式，最简单的是设置 Router 为 null（无 Router），这样会使用直线连接，其他连接方式包括折线、具有控制点的折线等，若想控制连接线不互相交叉也需要在 Router 中做文章。

19.3.2　连线的实现

Anchor 控制连接线端点在图形上的位置，即“锚点”的位置，最易于使用的是 ChopBoxAnchor，它先假设图形中心为连接点，然后计算这条假想连线与图形边缘的交汇点作为实际的锚点，其他 Anchor，还有 EllipseAnchor、LabelAnchor 和 XYAnchor 等。

Locator 的作用是定位图形，例如希望在连接线中点处以一个标签显示此连线的名称，就可以使用 MidpointLocator 来帮助定位这个标签，其他 Locator 还有 ArrowLocator 用于定位可旋转的修饰（Decoration，例如 PolygonDecoration）、BendpointLocator 用于定位连接控制点、ConnectionEndpointLocator 用于定位连接端点（通过指定 uDistance 和 vDistance 属性的值可以设置以端点为原点的坐标）。

实现简单图形的连接只要设置好连线对应的锚点即可，如例程 19-3 所示。

例程 19-3　Connection.java

```
package com.test.demo;

import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Display;
import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;

public class Connection {

public static void main(String args[]){
     Shell shell = new Shell();
     shell.setSize(350,350);
     shell.open();
     shell.setText("Demo 4");
     LightweightSystem lws = new LightweightSystem(shell);
     IFigure panel = new Figure();
     lws.setContents(panel);
     //创建两个四边形的图形实例
```

```
        RectangleFigure
            node1 = new RectangleFigure(),
            node2 = new RectangleFigure();
        //设置node1的背景色
        node1.setBackgroundColor(ColorConstants.red);
        //设置node1的大小和位置
        node1.setBounds(new Rectangle(30, 30, 64, 36));

        //设置node2的背景色
        node2.setBackgroundColor(ColorConstants.blue);
        //设置node2的大小和位置
        node2.setBounds(new Rectangle(100, 100, 64, 36));

        //创建一个连线的实例
        PolylineConnection conn = new PolylineConnection();
        //设置连线起点的锚点
        conn.setSourceAnchor(new ChopboxAnchor(node1));
        //设置连线目标的锚点
        conn.setTargetAnchor(new ChopboxAnchor(node2));
        //设置连线目标的装饰器
        conn.setTargetDecoration(new PolygonDecoration());

        Label label = new Label("Midpoint");
        label.setOpaque(true);
        label.setBackgroundColor(ColorConstants.buttonLightest);
        label.setBorder(new LineBorder());

        //添加连线的Locator
        conn.add(label, new MidpointLocator(conn, 0));

        //在底层Figure中添加了Figure
        panel.add(node1);
        panel.add(node2);
        panel.add(conn);
        //添加node1拖动的监听器
        new Dragger(node1);
        //添加node2拖动的监听器
        new Dragger(node2);
        Display display = Display.getDefault();
        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep ();
    }

    }
    static class Dragger extends MouseMotionListener.Stub implements MouseListener {
        public Dragger(IFigure figure){
            figure.addMouseMotionListener(this);
            figure.addMouseListener(this);
        }
        Point last;
```

```
        public void mouseReleased(MouseEvent e){}
        public void mouseClicked(MouseEvent e){}
        public void mouseDoubleClicked(MouseEvent e){}
        public void mousePressed(MouseEvent e){
            last = e.getLocation();
        }
        public void mouseDragged(MouseEvent e){
            Point p = e.getLocation();
            Dimension delta = p.getDifference(last);
            last = p;
            Figure f = ((Figure)e.getSource());
            //设置拖动的 Figure 的位置
            f.setBounds(f.getBounds().getTranslated(delta.width, delta.height));
        }
    };

    }
```

上例中，通过 PolylineConnection 建立了 node1 和 node2 的连线，并且设置了 node1 和 node2 拖动的监听器，用户可以拖动图形到窗口的任意位置，如图 19-5 所示。

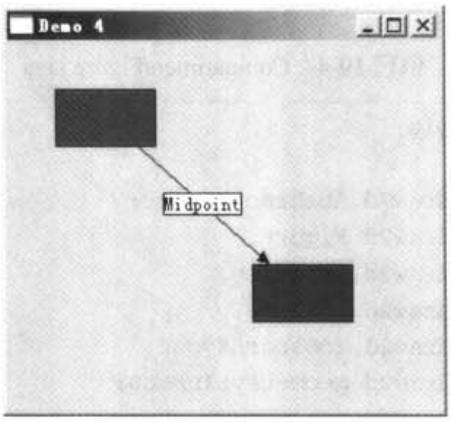


图 19-5　Draw2d 的连线

19.4　UML 关系图实现

Draw2d 可以通过图形和连线表现图的关系，本节将用实例介绍如何通过图形和连线绘制 UML 的关系图。

19.4.1　UML 关系图实现的要求

表关系实例中要包括表和列的 UML 图形(Figure)，其中表和列又包含属性和方法子图形。表和列之间要建立一个一对多的关系连线，如图 19-6 所示。

从图 19-6 中可以看出，实例应该建立一个 UML 的 Class 图形，另外，Class 图形中包括属性和方法的子图形，UML 的 Class 图形之间通过连线连接。

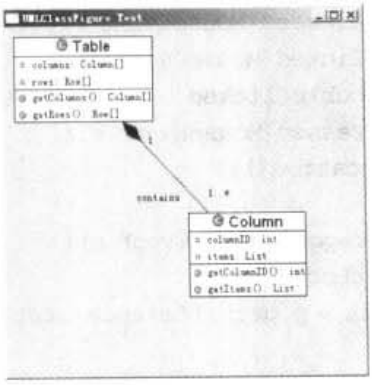


图 19-6　UML 关系图

19.4.2　CompartmentFigure 实现

CompartmentFigure 是方法和属性的图形，如图 19-6 所示。UML 关系图中有四个 CompartmentFigure 图形，即每个 UML 的 Class 都包含两个 CompartmentFigure 图形。另外，图形还带边框，每一项属性和方法都有图标关联（图标是通过 Label 设置的）。CompartmentFigure 的代码如例程 19-4 所示。

例程 19-4　CompartmentFigure.java

```
package com.test.table;

import org.eclipse.draw2d.AbstractBorder;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.ToolbarLayout;
import org.eclipse.draw2d.geometry.Insets;

public class CompartmentFigure extends Figure {

    public CompartmentFigure() {
      ToolbarLayout layout = new ToolbarLayout();
      layout.setMinorAlignment(ToolbarLayout.ALIGN_TOPLEFT);
      layout.setStretchMinorAxis(false);
      //设置子图形的间距
      layout.setSpacing(2);
      //设置布局管理器
      setLayoutManager(layout);
      //设置图形的边框
      setBorder(new CompartmentFigureBorder());
    }

    //图形的边框类
    public class CompartmentFigureBorder extends AbstractBorder {
      public Insets getInsets(IFigure figure) {
        return new Insets(1,0,0,0);
      }
```

```
        //重画图形的边框
        public void paint(IFigure figure, Graphics graphics, Insets insets) {
          graphics.drawLine(getPaintRectangle(figure, insets).getTopLeft(),
                        tempRect.getTopRight());
        }
      }
    }
```

19.4.3 UMLClassFigure 实现

UMLClassFigure 是 UML Class 的图形，如图 19-6 所示，UML 关系图中有两个 UMLClassFigure 图形，其中每个 UMLClassFigure 中又包含两个 CompartmentFigure。UMLClassFigure 的代码如例程 19-5 所示。

例程 19-5 UMLClassFigure.java

```
package com.test.table;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.LineBorder;
import org.eclipse.draw2d.ToolbarLayout;
import org.eclipse.swt.graphics.Color;

public class UMLClassFigure extends Figure {
      public static Color classColor = new Color(null,255,255,206);
      //属性子图形
      private CompartmentFigure attributeFigure = new CompartmentFigure();
      //方法子图形
      private CompartmentFigure methodFigure = new CompartmentFigure();
      public UMLClassFigure(Label name) {
        ToolbarLayout layout = new ToolbarLayout();
        //设置布局管理器
        setLayoutManager(layout);
        //设置图形的边框
        setBorder(new LineBorder(ColorConstants.black,1));
        //设置背景色
        setBackgroundColor(classColor);
        //设置图形是否透明
        setOpaque(true);

        //添加名字的标签
        add(name);
        //添加属性的图形
        add(attributeFigure);
        //添加方法的图形
        add(methodFigure);
      }
      public CompartmentFigure getAttributesCompartment() {
```

```
        return attributeFigure;
      }
      public CompartmentFigure getMethodsCompartment() {
        return methodFigure;
      }
    }
```

19.4.4 连线及测试类

如图 19-6 所示，连线中添加了装饰器用来表示一对多的菱形，以及用来描述关系的三个 Locator（“1..*”、“1”和“contains”），另外每个标签都设置了相应的显示图标，代码如例程 19-6 所示。

例程 19-6 UMLClassFigureTest.java

```
package com.test.table;

import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.PointList;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class UMLClassFigureTest {
 public static void main(String args[]){
    Display d = new Display();
    final Shell shell = new Shell(d);
    shell.setSize(400, 400);
    shell.setText("UMLClassFigure Test");
    LightweightSystem lws = new LightweightSystem(shell);
    //新建底层的图形
    Figure contents = new Figure();
    //设置底层图的布局方式为 XYLayout
    XYLayout contentsLayout = new XYLayout();
    contents.setLayoutManager(contentsLayout);

    Font classFont = new Font(null, "Arial", 12, SWT.BOLD);
    //添加表的显示文本及显示的图标
    Label classLabel1 = new Label("Table", new Image(d,
        UMLClassFigureTest.class.getResourceAsStream("class_obj.gif")));
    classLabel1.setFont(classFont);

    Label classLabel2 = new Label("Column", new Image(d,
        UMLClassFigureTest.class.getResourceAsStream("class_obj.gif")));
    classLabel2.setFont(classFont);

    //新建表和列的图形
```

```
final UMLClassFigure classFigure = new UMLClassFigure(classLabel1);
final UMLClassFigure classFigure2 = new UMLClassFigure(classLabel2);

//添加属性的显示文本及显示图标
Label attribute1 = new Label("columns: Column[]", new Image(d,
    UMLClassFigure.class.getResourceAsStream("field_private_obj.gif")));
Label attribute2 = new Label("rows: Row[]", new Image(d,
    UMLClassFigure.class.getResourceAsStream("field_private_obj.gif")));
Label attribute3 = new Label("columnID: int", new Image(d,
    UMLClassFigure.class.getResourceAsStream("field_private_obj.gif")));
Label attribute4 = new Label("items: List", new Image(d,
    UMLClassFigure.class.getResourceAsStream("field_private_obj.gif")));

//添加方法和属性的标签
classFigure.getAttributesCompartment().add(attribute1);
classFigure.getAttributesCompartment().add(attribute2);
classFigure2.getAttributesCompartment().add(attribute3);
classFigure2.getAttributesCompartment().add(attribute4);

Label method1 = new Label("getColumns(): Column[]", new Image(d,
    UMLClassFigure.class.getResourceAsStream("methpub_obj.gif")));
Label method2 = new Label("getRows(): Row[]", new Image(d,
    UMLClassFigure.class.getResourceAsStream("methpub_obj.gif")));
Label method3 = new Label("getColumnID(): int", new Image(d,
    UMLClassFigure.class.getResourceAsStream("methpub_obj.gif")));
Label method4 = new Label("getItems(): List", new Image(d,
    UMLClassFigure.class.getResourceAsStream("methpub_obj.gif")));

classFigure.getMethodsCompartment().add(method1);
classFigure.getMethodsCompartment().add(method2);
classFigure2.getMethodsCompartment().add(method3);
classFigure2.getMethodsCompartment().add(method4);

contentsLayout.setConstraint(classFigure, new Rectangle(10,10,-1,-1));
contentsLayout.setConstraint(classFigure2, new Rectangle(200, 200, -1, -1));

//新建连线
PolylineConnection c = new PolylineConnection();
//添加图形的锚点
ChopboxAnchor sourceAnchor = new ChopboxAnchor(classFigure);
ChopboxAnchor targetAnchor = new ChopboxAnchor(classFigure2);
c.setSourceAnchor(sourceAnchor);
c.setTargetAnchor(targetAnchor);

//添加连线的装饰器
PolygonDecoration decoration = new PolygonDecoration();
PointList decorationPointList = new PointList();
decorationPointList.addPoint(0,0);
decorationPointList.addPoint(-2,2);
decorationPointList.addPoint(-4,0);
decorationPointList.addPoint(-2,-2);
```

```
        decoration.setTemplate(decorationPointList);
        c.setSourceDecoration(decoration);

        //添加连线的Locator
        ConnectionEndpointLocator targetEndpointLocator =
              new ConnectionEndpointLocator(c, true);
        targetEndpointLocator.setVDistance(15);
        Label targetMultiplicityLabel = new Label("1..*");
        c.add(targetMultiplicityLabel, targetEndpointLocator);
        //添加连线的Locator
        ConnectionEndpointLocator sourceEndpointLocator =
            new ConnectionEndpointLocator(c, false);
        sourceEndpointLocator.setVDistance(15);
        Label sourceMultiplicityLabel = new Label("1");
        c.add(sourceMultiplicityLabel, sourceEndpointLocator);
        //添加连线的Locator
        ConnectionEndpointLocator relationshipLocator =
            new ConnectionEndpointLocator(c,true);
        relationshipLocator.setUDistance(10);
        relationshipLocator.setVDistance(-20);
        Label relationshipLabel = new Label("contains");
        c.add(relationshipLabel,relationshipLocator);

        //把表、列和连线（即关系）添加到底层图形上
        contents.add(classFigure);
        contents.add(classFigure2);
        contents.add(c);

        lws.setContents(contents);
        shell.open();
        while (!shell.isDisposed())
            while (!d.readAndDispatch())
                d.sleep();
    }
}
```

通过上面的 CompartmentFigure、UMLClassFigure 和测试类 UMLClassFigureTest 实现了如图 19-6 所示的 UML 类关系图，读者可以通过本书源代码运行测试类。另外，要运行 Draw2d 的应用，应该添加 Draw2d 实现的 Jar 包才能运行。

19.5 小　　结

Draw2d 是基于 SWT 的轻量级组件系统，它是实现 GEF 视图的基础。本章只能概括性地介绍 Draw2d 的一些概念及实现，读者可以参考 Draw2d 的联机帮助文档。

通过学习本章读者应该了解如下两方面的内容。

- 图形（Figure）
- 连线（Connection）

第 20 章　GEF 原理及实现

GEF（Graphical Editor Framework）是一个功能强大的可视化模型编辑框架，通过 GEF，Eclipse 的开发者能够快速开发出图形编辑器。

本章将介绍 GEF 中的几个概念及其实现，包括如下几个部分。

- GEF 概述
- 模型（Model）
- 视图（View）
- 控制器（Control）
- 请求和编辑策略（Request and EditPolicies）
- 命令（commands）
- 模型同步
- 选项板（PaletteRoot）

20.1　GEF 概述

GEF 是一个图形化编辑框架，它允许开发人员以图形化的方式展示和编辑模型，从而提升用户体验。

20.1.1　GEF 简介

GEF 最早是 Eclipse 的一个内部项目，后来逐渐转变为 Eclipse 的一个开源工具项目，Eclipse 的不少其他子项目都需要它的支持。

GEF 的优势是提供了标准的 MVC（Model-View-Control）结构，即模型、视图和控制器（在此视图和插件的视图是两个完全不同的概念）。开发人员可以利用 GEF 来完成以上这些功能，而不需要自己重新设计。与其他一些 MVC 编辑框架相比，GEF 的一个主要设计目标是尽量减少模型和视图之间的依赖，好处是可以根据需要选择任意模型和视图的组合，而不必受开发框架的局限。

20.1.2　GEF 工作原理

GEF 中模型和视图是两个完全没有联系的实体，两者都是通过控制器间接进行通信的。这样很好地解决了模型和视图之间的耦合，但是也增加了控制器的负担。GEF 通过请求和编辑策略及命令模式很好地解决了控制器的负担，使每一个部分保持最小的依赖关系。GEF 的命令请求及响应流程如下。

（1）GEF 框架接收用户的操作，并把操作转换成相应的请求。

（2）控制器接收请求并把请求交由相应的编辑策略管理器处理。

（3）编辑策略管理器根据请求的类型新建相应的命令处理请求。

（4）GEF 框架执行命令，命令负责对模型的修改。

（5）模型属性修改后，触发事件（控制器就是模型的监听器）。

（6）监听器（控制器就是模型的监听器）监听到事件后处理相应的操作，并通知视图进行修改。

（7）视图更新显示界面，使之与模型同步。

创建GEF编辑器，首先Eclipse创建控制器（EditorPart）设置好控制器的输入（IEditorInput），另外控制器将初始化 EditDomain、viewers 和 palette 等，如图 20-1 所示。

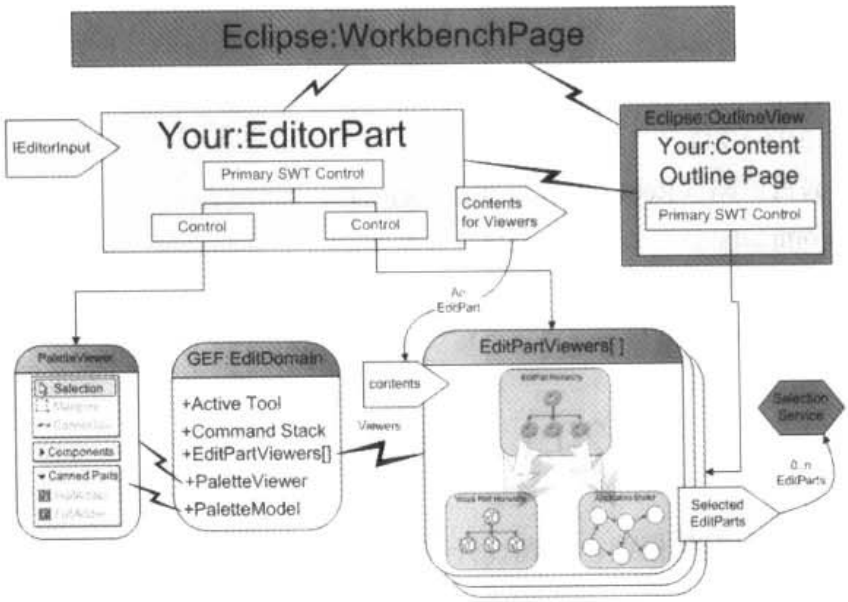


图 20-1　GEF 结构图

20.2　模　　型

GEF 的模型只与控制器打交道，而不知道任何与视图有关的东西，具体介绍如下。

20.2.1　模型的功能

在 GEF 框架中，模型（Model）是非常简单的一部分，用户可把模型理解成一个简单的可持久化的实体。GEF 的模型只与控制器打交道，而不知道任何与视图有关的东西。

但为了能让控制器知道模型的变化，应该把控制器作为事件监听者注册在模型中，当模型发生变化时，就触发相应的事件给控制器，后者负责通知视图进行更新。

20.2.2　模型的实现

典型的模型对象会包含 PropertyChangeSupport 类型的成员变量，用来触发事件，通知监

听器即控制器。一般来说，模型会实现一个基类，用来封装事件机制，主要包括添加监听器、删除监听器，以及模型属性改变触发的相应事件，代码如例程 20-1 所示。

例程 20-1 Element.java

```
package com.example.model;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

public abstract class Element implements Cloneable, Serializable {

   PropertyChangeSupport listeners = new PropertyChangeSupport(this);

   //添加监听器
   public void addPropertyChangeListener(PropertyChangeListener l) {
      listeners.addPropertyChangeListener(l);
   }

   //触发属性改变的事件
   protected void firePropertyChange(String prop, Object old, Object newValue)
{
      listeners.firePropertyChange(prop, old, newValue);
   }

   //触发结构改变的事件
   protected void fireStructureChange(String prop, Object child) {
      listeners.firePropertyChange(prop, null, child);
   }

   //删除监听器
   public void removePropertyChangeListener(PropertyChangeListener l) {
      listeners.removePropertyChangeListener(l);
   }

}
```

另外，当用户希望通过属性视图编辑模型属性时，模型要实现 IPropertySource 接口，IPropertySource 接口的方法解释如下。

（1）getEditableValue：方法定义为“public Object getEditableValue();”，表示得到在属性页中能编辑的值，可以返回 this 表示当前模型。

（2）getPropertyDescriptors：方法定义为“public IPropertyDescriptor[] getPropertyDescriptors();”，表示得到 IPropertyDescriptor 类型的数组，其中每一项为属性页中能编辑的项。

（3）getPropertyValue：方法定义为“public Object getPropertyValue(Object id);”，表示通过 id 值得到某个属性，在添加每一项 IPropertyDescriptor 时都会指定 id 值。

（4）isPropertySet：方法定义为“public boolean isPropertySet(Object id);”，表示特定属性 id 值是否改变。

（5）setPropertyValue：方法定义为“public void setPropertyValue(Object id, Object value);”，表示通过 id 和值设置某一项属性的值。

（6）resetPropertyValue：方法定义为“public void resetPropertyValue(Object id);”，通过 id 重置属性。

如果模型和其他模型有关系（在视图上为连线），模型要维护这些关系，并适当地持久化，当模型的属性修改后，模型要触发相应的事件通知监听器。一个简单的模型实现如例程 20-2 所示。

例程 20-2 Node.java

```
package com.example.model;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.draw2d.geometry.Point;
import org.eclipse.ui.views.properties.ComboBoxPropertyDescriptor;
import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.IPropertySource;
import org.eclipse.ui.views.properties.TextPropertyDescriptor;

public class Node extends Element implements IPropertySource {

    //定义属性的常量 ID
   final public static String PROP_LOCATION = "LOCATION";
   final public static String PROP_NAME = "NAME";
   final public static String PROP_VISIBLE = "VISIBLE";
   final public static String PROP_INPUTS = "INPUTS";
   final public static String PROP_OUTPUTS = "OUTPUTS";

   protected Point location = new Point(0, 0);

   protected String name = "Node";

   protected boolean visible = true;

   //定义属性的编辑项
   protected IPropertyDescriptor[] descriptors = new IPropertyDescriptor[] {
         new TextPropertyDescriptor(PROP_NAME, "Name"),
         new ComboBoxPropertyDescriptor(PROP_VISIBLE, "Visible", new String[]
          { "true", "false" }) };

   //定义模型连线的列表
   protected List outputs = new ArrayList(5);
   protected List inputs = new ArrayList(5);

   public void addInput(Connection connection) {
      this.inputs.add(connection);
      //当输入的连线改变后，触发结构改变的事件
      fireStructureChange(PROP_INPUTS, connection);
```

```
}

public void addOutput(Connection connection) {
    this.outputs.add(connection);
    //当输出的连线改变后，触发结构改变的事件
    fireStructureChange(PROP_OUTPUTS, connection);
}

public List getIncomingConnections() {
    return this.inputs;
}

public List getOutgoingConnections() {
    return this.outputs;
}

public void removeInput(Connection connection) {
    this.inputs.remove(connection);
    fireStructureChange(PROP_INPUTS, connection);
}

public void removeOutput(Connection connection) {
    this.outputs.remove(connection);
    fireStructureChange(PROP_OUTPUTS, connection);
}

public boolean isVisible() {
    return visible;
}

public void setVisible(boolean visible) {
    if (this.visible == visible) {
        return;
    }
    this.visible = visible;
    firePropertyChange(PROP_VISIBLE, null, Boolean.valueOf(visible));
}

public String getName() {
    return name;
}

public void setName(String name) {
    if (this.name.equals(name)) {
        return;
    }
    this.name = name;
    //当模型名字改变后，通知监听器更新名字的显示
    firePropertyChange(PROP_NAME, null, name);
}
```

```
    public void setLocation(Point p) {
        if (this.location.equals(p)) {
            return;
        }
        this.location = p;
        //当模型位置改变后，通知监听器更新模型的位置
        firePropertyChange(PROP_LOCATION, null, p);
    }

    public Point getLocation() {
        return location;
    }

    public Object getEditableValue() {
        return this;
    }

    //得到属性编辑项
    public IPropertyDescriptor[] getPropertyDescriptors() {
        return descriptors;
    }

    //得到属性编辑项的值
    public Object getPropertyValue(Object id) {
        if (PROP_NAME.equals(id))
            return getName();
        if (PROP_VISIBLE.equals(id))
            return isVisible() ? new Integer(0) : new Integer(1);
        return null;
    }

    public boolean isPropertySet(Object id) {
        return true;
    }

    public void resetPropertyValue(Object id) {

    }

    //设置属性的值
    public void setPropertyValue(Object id, Object value) {
        if (PROP_NAME.equals(id))
            setName((String) value);
        if (PROP_VISIBLE.equals(id))
            setVisible(((Integer) value).intValue() == 0);
    }
}
```

上例中实现了模型对应的属性视图，另外当属性被修改后，模型将会调用“firc...”方法触发相应的模型改变事件。

20.3 视 图

GEF 视图通常利用 Draw2d 图形（IFigure）作为表现方式，用户可以通过 Draw2d 的功能实现图形的展示、布局及图形的编辑。

20.3.1 视图的功能

视图（View）是模型图形方式的表现，它以图形的方式展示模型。视图的作用和 JFace 中的 Viewer 十分类似，而控制器（EditPart）就相当于是它的 ContentProvider 和 LabelProvider，通过 setContents()方法来指定视图的输入。

GEF 经常使用的 Editor 是一个 GraphicalEditorWithPalette（GEF 提供的 Editor，是 EditorPart 的子类，具有图形化编辑区域和一个工具条）的实例，这个 Editor 使用 GraphicalEditViewer 和 PaletteViewer 这两个视图类，PaletteViewer 也是 GraphicalEditViewer 的子类（表示选项板视图）。开发人员要在 configureGraphicalViewer()和 initializeGraphicalViewer()这两个方法里对 EditPartViewer 进行定制，包括指定它的 contents 和 EditPartFactory 等。

EditPartViewer 同时也是 ISelectionProvider，这样当用户在编辑区域做选择操作时，注册的 SelectionChangeListener 就可以收到选择事件。EditPartViewer 会维护各个 EditPart 的选中状态，如果没有被选中的 EditPart，则默认选中的是作为 contents 的 EditPart。

20.3.2 视图的实现

视图的实现相对模型来说较为复杂，除了模型的显示功能以外，还要提供编辑功能、回显（Feedback）、工具提示（ToolTip）等。

在 GEF 中，通常视图是利用 Draw2d 图形（IFigure）作为表现方式，用户可以通过 Draw2d 的功能实现图形的展示、布局及图形的编辑。例程 20-3 显示的是一个带边框的视图的实现。

例程 20-3 KCGContentFigure.java

```
package com.free.table.editor.figure;

import org.eclipse.draw2d.AbstractBorder;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.ToolbarLayout;
import org.eclipse.draw2d.geometry.Insets;
import org.eclipse.swt.graphics.Color;

public class KCGContentFigure extends Figure {

    public static Color classColor = new Color(null, 255, 255, 206);

    public KCGContentFigure() {
```

```
        //指定视图的布局
        FillLayout layout = new FillLayout();
        layout.setMinorAlignment(ToolbarLayout.ALIGN_TOPLEFT);
        layout.setStretchMinorAxis(false);
        setLayoutManager(layout);
        //设置图形的边框
        setBorder(new CompartmentFigureBorder());
    }

    public class CompartmentFigureBorder extends AbstractBorder {
        public Insets getInsets(IFigure figure) {
            return new Insets(1, 0, 0, 0);
        }

        //重画图形的边框
        public void paint(IFigure figure, Graphics graphics, Insets insets) {
            graphics.drawLine(getPaintRectangle(figure, insets).getTopLeft(),
                    tempRect.getTopRight());
        }
    }

}
```

20.4 控 制 器

控制器是 GEF 框架的核心，它负责模型和视图之间的通信，具体介绍如下。

20.4.1 控制器的功能

在 GEF 的 MVC 结构里，控制器（Control）是模型与视图之间的桥梁，也是整个 GEF 的核心。它不仅要监听模型的变化，当用户编辑视图时，还要把编辑结果反映到模型上。

在 GEF 中，控制器是由一组 EditPart 对象共同组成的，每一个模型对象都对应一个 EditPart 对象。应用程序中需要有一个 EditPartFactory 对象负责根据给定模型对象创建对应的 EditPart 对象，这个工厂类将在创建模型时被调用。

控制器是 GEF 中最复杂的一部分，GEF 把控制器完成的工作又分成了几个部分，包括请求和编辑策略及引申出来的命令模式，如图 20-2 所示。

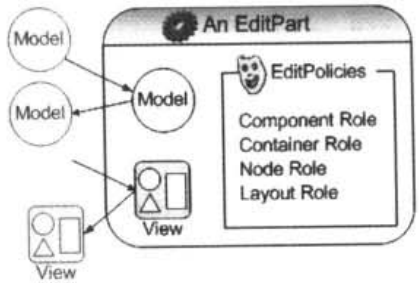


图 20-2 控制器

用户的编辑操作被转换为一系列请求（Request），Eclipse 中有很多种类的请求，这些种类在 GEF 里被称为角色（Role）。在 GEF 里有图形化和非图形化这两大类角色，前者如“Layout Role”对应和布局有关的操作，后者如“Connection Role”对应和连接有关的操作等。角色这个概念是通过编辑策略（EditPolicy）来实现的，EditPolicy 的主要功能是根据请求创建相应的命令（Command），而后者会直接操作模型对象。

对每一个 EditPart，用户都可以“安装”一些 EditPolicy。用户对这个 EditPart 的特定操作会被交给已安装的对应 EditPolicy 处理。这样做的直接好处是可以在不同 EditPart 之间共享一些重复操作。

20.4.2 控制器的实现

每一个模型都会对应一个控制器，控制器主要负责模型和视图的同步。控制器要监听模型的属性改变的事件，并通知视图更新。另外，控制器还要设置相应的编辑策略，用来处理由视图接收的请求。一个简单的控制器代码如例程 20-4 所示。

例程 20-4 NodePart.java

```
package com.example.parts;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.List;

import org.eclipse.draw2d.ChopboxAnchor;
import org.eclipse.draw2d.ConnectionAnchor;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.geometry.Dimension;
import org.eclipse.draw2d.geometry.Point;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.ConnectionEditPart;
import org.eclipse.gef.EditPolicy;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.NodeEditPart;
import org.eclipse.gef.Request;
import org.eclipse.gef.RequestConstants;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;
import org.eclipse.gef.tools.DirectEditManager;
import org.eclipse.jface.viewers.TextCellEditor;

import com.example.figures.NodeFigure;
import com.example.model.Node;
import com.example.policies.NodeDirectEditPolicy;
import com.example.policies.NodeEditPolicy;
import com.example.policies.NodeGraphicalNodeEditPolicy;

public class NodePart extends AbstractGraphicalEditPart implements
PropertyChangeListener, NodeEditPart {
```

```
protected DirectEditManager manager;

//处理请求
public void performRequest(Request req) {
   if (req.getType().equals(RequestConstants.REQ_DIRECT_EDIT)) {
      if (manager == null) {
         NodeFigure figure = (NodeFigure) getFigure();
         manager = new NodeDirectEditManager(this,
               TextCellEditor.class, new NodeCellEditorLocator(figure));
      }
      manager.show();
   }
}

//监听模型属性改变
public void propertyChange(PropertyChangeEvent evt) {
   if (evt.getPropertyName().equals(Node.PROP_LOCATION))
      refreshVisuals();
   else if (evt.getPropertyName().equals(Node.PROP_NAME))
      refreshVisuals();
   else if (evt.getPropertyName().equals(Node.PROP_INPUTS))
      refreshTargetConnections();
   else if (evt.getPropertyName().equals(Node.PROP_OUTPUTS))
      refreshSourceConnections();
}

//创建模型对应的视图
protected IFigure createFigure() {
   return new NodeFigure();
}

//设置编辑策略
protected void createEditPolicies() {
   installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new NodeDirectEditPolicy());
   installEditPolicy(EditPolicy.COMPONENT_ROLE, new NodeEditPolicy());
   installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new
   NodeGraphicalNodeEditPolicy());
}

//注册自己成为模型的监听器
public void activate() {
   if (isActive()) {
      return;
   }
   super.activate();
   ((Node) getModel()).addPropertyChangeListener(this);
}

//把自己从模型的监听器中删除
public void deactivate() {
   if (!isActive()) {
```

```
            return;
        }
        super.deactivate();
        ((Node) getModel()).removePropertyChangeListener(this);
    }

    //更新视图的显示
    protected void refreshVisuals() {
        Node node = (Node) getModel();
        Point loc = node.getLocation();
        Dimension size = new Dimension(150, 40);
        Rectangle rectangle = new Rectangle(loc, size);
        ((NodeFigure) this.getFigure()).setName(((Node) this.getModel()).getName());
        ((GraphicalEditPart) getParent()).setLayoutConstraint(this, getFigure(),
         rectangle);
    }

    //得到源连接的连接点
    public ConnectionAnchor getSourceConnectionAnchor(ConnectionEditPart connection) {
        return new ChopboxAnchor(getFigure());
    }

    public ConnectionAnchor getSourceConnectionAnchor(Request request) {
        return new ChopboxAnchor(getFigure());
    }
    //得到目标连接的连接点
    public ConnectionAnchor getTargetConnectionAnchor(ConnectionEditPart connection) {
        return new ChopboxAnchor(getFigure());
    }

    public ConnectionAnchor getTargetConnectionAnchor(Request request) {
        return new ChopboxAnchor(getFigure());
    }

    //得到以模型作为源的连接列表
    protected List getModelSourceConnections() {
        return ((Node) this.getModel()).getOutgoingConnections();
    }

    //得到以模型作为目标的连接列表
    protected List getModelTargetConnections() {
        return ((Node) this.getModel()).getIncomingConnections();
    }

}
```

上例通过继承 AbstractGraphicalEditPart 类实现了一个 EditPart，另外，当 GEF 框架创建一个模型时，都会为此模型创建一个对应的控制器。创建控制器的操作是在控制器工厂中完成的，用户可以在编辑器初始化时指定控制器工厂，例如“getGraphicalViewer().setEditPartFactory (new PartFactory());”，其中 PartFactory 为控制器工厂，代码如例程 20-5 所示。

例程 20-5 PartFactory.java

```
package com.example.parts;

import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPartFactory;

import com.example.model.Connection;
import com.example.model.Diagram;

public class PartFactory implements EditPartFactory {

   public EditPart createEditPart(EditPart context, Object model) {
      EditPart part = null;
      //根据模型创建相应的 EditPart
      if (model instanceof Diagram)
         part = new DiagramPart();
      else if (model instanceof Connection)
         part = new ConnectionPart();
      else
         part = new NodePart();
      //设置 EditPart 对应的模型
      part.setModel(model);
      //返回根据模型创建的 EditPart
      return part;
   }
}
```

通过创建控制器和控制器工厂，GEF 编辑器将会知道在模型创建后，自动调用控制器工厂创建对应的控制器，从而实现模型和控制器的一一对应关系。

20.5 请求和编辑策略

请求和编辑策略是 GEF 框架中减轻控制器的负担、减小代码耦合度而实现的一种解决方案，具体实现如下。

20.5.1 请求和编辑策略（Request and EditPolicies）

请求和编辑策略对初学者来说是比较难理解的部分，但正是因为这种机制才使得 GEF 框架功能强大，而且非常灵活。想象一下，如果用户所有操作的响应都由 EditPart 来处理会怎样，如果这样的话，EditPart 的代码将会“又臭又长”，这是让人无法忍受的。

在 EditPart 中，可以通过设置不同的编辑策略（EditPolicies）来处理不同的请求，这样，一方面，可以把代码从 EditPart 中解放出来，分别由不同的 EditPolicies 进行处理，另一方面，用户可以着力于自己的关注点，但由此也增加读者学习 GEF 框架的时间。

另外，在 EditPart 中设置编辑策略时，要指定相应的角色（Role），角色只是一个标识，在同一个 EditPart 中不能存在两个相同角色的编辑策略，读者可以在 GEF 的联机文档中找到

详细的编辑策略、请求和角色说明。

20.5.2 编辑策略的实现

控制器中通过 createEditPolicies 方法添加编辑策略，每种编辑策略负责处理相应的请求。通常请求一般会对模型进行操作，在 EditPolicies 中，可以通过命令的方式操作模型，命令将在后面介绍。EditPolicies 的示例代码如例程 20-6 所示。

例程 20-6 DiagramLayoutEditPolicy.java

```
package com.example.policies;

import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.Request;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.XYLayoutEditPolicy;
import org.eclipse.gef.requests.CreateRequest;

import com.example.commands.CreateNodeCommand;
import com.example.commands.MoveNodeCommand;
import com.example.model.Diagram;
import com.example.model.Node;
import com.example.parts.NodePart;

public class DiagramLayoutEditPolicy extends XYLayoutEditPolicy {

    protected Command createAddCommand(EditPart child, Object constraint) {
        return null;
    }

    //创建模型位置改变的命令
    protected Command createChangeConstraintCommand(EditPart child, Object constraint) {
     //如果位置改变的不是 Node 则返回
        if (!(child instanceof NodePart))
            return null;
        if (!(constraint instanceof Rectangle))
            return null;

        MoveNodeCommand cmd = new MoveNodeCommand();
        cmd.setNode((Node) child.getModel());
        //设置模型新的位置信息
        cmd.setLocation(((Rectangle) constraint).getLocation());
        return cmd;

    }

    //获得创建模型的命令
    protected Command getCreateCommand(CreateRequest request) {
     //判断请求创建的是否为 Node
```

```
            if (request.getNewObject() instanceof Node) {
               //新建CreateNodeCommand
               CreateNodeCommand cmd = new CreateNodeCommand();
               //设置父模型
               cmd.setDiagram((Diagram) getHost().getModel());
               //设置当前模型
               cmd.setNode((Node) request.getNewObject());
               Rectangle constraint = (Rectangle) getConstraintFor(request);
               //设置模型的位置信息
               cmd.setLocation(constraint.getLocation());
               //返回Command对象
               return cmd;
            }
            return null;
         }

         protected Command getDeleteDependantCommand(Request request) {
            return null;
         }
      }
```

通过实现此编辑策略，GEF 编辑器将能够处理 XYLayoutEditPolicy 所能响应的相关请求，并交由相应的 Command 进行处理。

20.6 命 令

为了提高代码的重用性，以及实现 GEF 编辑器的功能，GEF 采用命令模式提供了对模型的操作方式。

20.6.1 命令（Commands）的功能

在 EditPolicies 中新建了命令，如果返回的命令不为空，GEF 框架会执行命令，对模型进行相应的修改。

Command 是 GEF 中的一个抽象类，主要实现如下几个方法。

（1）execute：这是命令的执行方法，当请求结束并获得 Command 后，GEF 框架会负责执行此方法。

（2）undo：对模型修改后，可以通过 undo 进行撤销。

（3）redo：当用户撤销后，能通过 redo 重复上一次的操作。

20.6.2 命令的实现

GEF 通过一个命令堆栈（CommandStack）保存 Command 对象。用户可以通过命令堆栈撤销或重复对模型所做的操作。下面代码演示模型位置改变的命令，如例程 20-7 所示。

例程 20-7 MoveNodeCommand.java

```
package com.example.commands;

import org.eclipse.draw2d.geometry.Point;
import org.eclipse.gef.commands.Command;

import com.example.model.Node;

public class MoveNodeCommand extends Command {
    //命令对应的模型
    private Node node;

    //原始位置
    private Point oldPos;

    //新位置
    private Point newPos;

    //设置新位置
    public void setLocation(Point p) {
        this.newPos = p;
    }

    //设置命令对应的模型
    public void setNode(Node node) {
        this.node = node;
    }

    //执行命令
    public void execute() {
        //保存模型原来的位置
        oldPos = this.node.getLocation();
        node.setLocation(newPos);
    }

    public String getLabel() {
        return "Move Node";
    }

    //重复上一操作
    public void redo() {
        this.node.setLocation(newPos);
    }

    //撤销设置新位置的操作
    public void undo() {
        this.node.setLocation(oldPos);
    }
}
```

通过命令堆栈（CommandStack），GEF 编辑器能轻松完成用户的撤销和重做操作，GEF 框架会维护命令堆栈。

20.7 模 型 同 步

为了保证数据和展现的同步，必须要有一种消息通知机制，当模型被修改时通知相应的视图进行更新，本节将介绍如何实现这种消息机制。

20.7.1 模型同步的功能

模型是 GEF 中的实体，当模型的属性被修改后，控制器会监听到模型属性改变的事件，然后通知视图进行更新，模型和视图之间没有直接的联系，如图 20-3 所示。

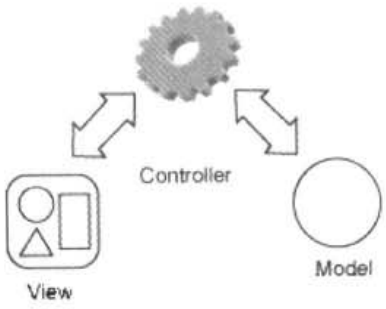


图 20-3 模型、控制器和视图的关系

要通过事件的通知机制获得模型同步的步骤如下。

（1）模型中包含注册监听器的机制，本书的范例通过 PropertyChangeSupport 和 PropertyChangeListener 机制添加模型监听器。

（2）把控制器注册为模型监听器，在此通过控制器实现 PropertyChangeListener 接口，并把自己添加到模型的监听器即可。

（3）当模型触发属性改变的事件后，控制器监听到事件并相应地更新视图。

另外，并不是模型的所有属性的修改都必须通知控制器，一般情况下，当模型的属性更改后要引起视图的更新时，必须触发事件通知控制器，因为模型和视图之间没有任何的关联。

20.7.2 模型同步的实现

典型的模型对象会包含 PropertyChangeSupport 类型的成员变量，用来触发事件，通知监听器即控制器。当模型的属性被修改后，控制器将更新视图的显示。例如，某个节点的名字修改后，将要触发相应的事件，通知控制器，如下代码如示。

```
public void setName(String name) {
   if (this.name.equals(name)) {
      return;
   }
   this.name = name;
```

```
        //当模型名字改变后，通知监听器更新名字的显示
        firePropertyChange(PROP_NAME, null, name);
    }
```

当模型的名字改变后，将通过 firePropertyChange 方法触发模型名字被修改的事件，其中第一个参数为事件类型标识（唯一的标识），表示模型的什么属性被改变，第二个参数为改变前的值，第三个参数为改变后的新值。

模型触发事件后，控制器将监听到模型属性修改的事件，通过事件的类型，即 firePropertyChange 方法的第一个参数判断是哪一个属性被修改，然后执行相应的操作，如下代码所示。

```
    //监听模型属性改变
    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(Node.PROP_LOCATION))
         //更新视图
           refreshVisuals();
        else if (evt.getPropertyName().equals(Node.PROP_NAME))
           refreshVisuals();
        else if (evt.getPropertyName().equals(Node.PROP_INPUTS))
         //更新连接
           refreshTargetConnections();
        else if (evt.getPropertyName().equals(Node.PROP_OUTPUTS))
           refreshSourceConnections();
    }
```

当控制器监听到模型的改变事件后，将会通过视图的更新，从而实现了模型数据和视图显示的同步。

20.8　选　项　板

选项板提供了用户选择模型元素，并提供相应的机制把模型和视图加入图形编辑器的能力。

20.8.1　选项板（PaletteRoot）的功能

GEF 是一个图形编辑框架，它的主编辑区域是一个图形的视图（类似 JFace 的 Viewer），另外，用户还可以选择选项板上相应的模型，把选择的模型对应的视图加入到图形编辑区域进行编辑。

在 GEF 中，图形编辑器实现类的 getPaletteRoot 方法将返回一个 PaletteRoot 对象，用户可以在 PaletteRoot 中添加模型选项入口项（ToolEntry）。例如，在选择板中加入一个连线模型的入口，可以加入“new ConnectionCreationToolEntry("Connection", "Create a connection", null, null, null);”，ToolEntry 的参数描述了选项板中选项的显示标签、显示图标、提示信息、模型的模板类及模型的创建工厂类等信息。

20.8.2 选项板的实现

在选项板中创建模型的入口，能添加模型的选择项（ToolEntry）和模型分组项（PaletteDrawer），代码如例程 20-8 所示。

例程 20-8 PaletteFactory.java

```
package com.example.tools;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.gef.palette.CombinedTemplateCreationEntry;
import org.eclipse.gef.palette.ConnectionCreationToolEntry;
import org.eclipse.gef.palette.PaletteContainer;
import org.eclipse.gef.palette.PaletteDrawer;
import org.eclipse.gef.palette.PaletteGroup;
import org.eclipse.gef.palette.PaletteRoot;
import org.eclipse.gef.palette.SelectionToolEntry;
import org.eclipse.gef.palette.ToolEntry;
import org.eclipse.gef.requests.SimpleFactory;

import com.example.model.Node;

public class PaletteFactory {

   public static PaletteRoot createPalette() {
      //新建选项板
      PaletteRoot paletteRoot = new PaletteRoot();
      //添加选项板中的模型选项
      paletteRoot.addAll(createCategories(paletteRoot));
      return paletteRoot;
   }

   private static List createCategories(PaletteRoot root) {
      List categories = new ArrayList();

      //添加分组选项
      categories.add(createControlGroup(root));
      categories.add(createComponentsDrawer());

      return categories;
   }

   private static PaletteContainer createControlGroup(PaletteRoot root) {
      PaletteGroup controlGroup = new PaletteGroup("Control Group");

      List entries = new ArrayList();
      //添加取消选择模型项
      ToolEntry tool = new SelectionToolEntry();
```

```
        entries.add(tool);
        //设置 SelectionToolEntry 为默认选项
        root.setDefaultEntry(tool);

        //添加连线模型选项入口
        tool = new ConnectionCreationToolEntry("Connection", "Create a connection",
         null, null, null);
        entries.add(tool);

        controlGroup.addAll(entries);
        return controlGroup;
    }

    private static PaletteContainer createComponentsDrawer() {

        //添加组件分组页
        PaletteDrawer drawer = new PaletteDrawer("Components");

        List entries = new ArrayList();
        //添加 Node 模型选项入口
        ToolEntry tool = new CombinedTemplateCreationEntry("Node", "Create a new
         Node", Node.class, new SimpleFactory(
                Node.class), null, null);
        entries.add(tool);

        drawer.addAll(entries);
        return drawer;
    }
}
```

上面代码加入了连线模型、分组项、取消选择项和一个节点项（Node 模型）。建立的选项板如图 20-4 所示。

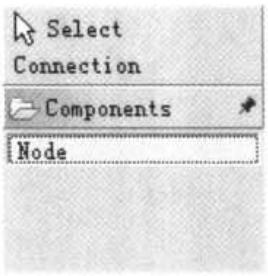


图 20-4　选项板

20.9　小　　结

本章介绍了 GEF 中的一些概念和元素，并通过相应的例子介绍了这些元素的使用方式和场景。

GEF 是一个完善的图形编辑框架，它提供了图形编辑的解决方案，用户可以通过它的机

制构造出功能非常强大的图形编辑器，但 GEF 还是比较复杂的框架，在提供了图形编辑的各个环节松耦合的灵活性之时，也带来了用户必须了解各环节实现的学习时间。下面的章节将通过一个完整的实例介绍如何构建一个 GEF 的模型编辑器。

另外，用户还可以在 GEF 的联机帮助文档中查找 GEF 的详细资料。通过学习本章读者应该了解如下几个方面的概念和内容。

- 模型（Model）
- 视图（View）
- 控制器（Control）
- 请求和编辑策略（Request and EditPolicies）
- 命令（Commands）
- 模型同步
- 选项板（PaletteRoot）

第 21 章　GEF 表模型编辑器的实现

GEF 表模型编辑器充分运用了本书介绍的基本知识，它实现了一个关系表模型的编辑器，更加贴近于实际应用。本章将介绍表模型编辑器应用的软件结构、GEF 编辑器的每一个功能，以及一些其他的应用。

本章内容分为如下几个部分。

- 程序的软件结构
- GEF 编辑器的实现
- 模型的实现
- 视图的实现
- 控制器的实现
- Factory 的实现
- EditPolicy 的实现
- Command 的实现
- 大纲的实现

21.1　程序的软件结构

表模型编辑器是基于 GEF 框架，采用 MVC 的设计模式开发的图形编辑器，主要包括模型、视图和控制器 3 部分。

21.1.1　实现功能

作为一个 GEF 的编辑器，程序要求实现 MVC 模式的 3 个部分。表模型编辑器主要实现如下的功能。

（1）编辑器：主要功能为初始化 GEF 编辑环境。

（2）模型：包括 Schema（即底层的模型 schema，其他的模型都直接或间接为底层模型的子模型）、Table（表模型）、Column（列模型）和 Relationship（连线模型）。

（3）视图：包括 SchemaFigure（底层视图）、TableFigure（表视图）和 ColumnsFigure（列视图），连线视图为 Draw2d 提供的连接视图 PolylineConnection。

（4）控制器：控制器和模型是一一对应的，包括 SchemaDiagramPart（底层控制器）、TablePart（表控制器）、ColumnPart（列控制器）和 RelationshipPart（连线控制器）。

（5）编辑策略：包括添加模型的编辑策略、删除模型的编辑策略、移动模型的编辑策略等。

（6）命令：编辑策略通过 Command 对模型进行编辑，包括添加模型的 Command、删除模型的 Command 和移动模型的 Command 等。

21.1.2 程序的界面要求

表模型编辑器要求能够在编辑器中加入表，以及在每一个表中能加入列，另外表和表之间能建立主外键的关系，并且能编辑主外键关系。其主界面如图 21-1 所示。

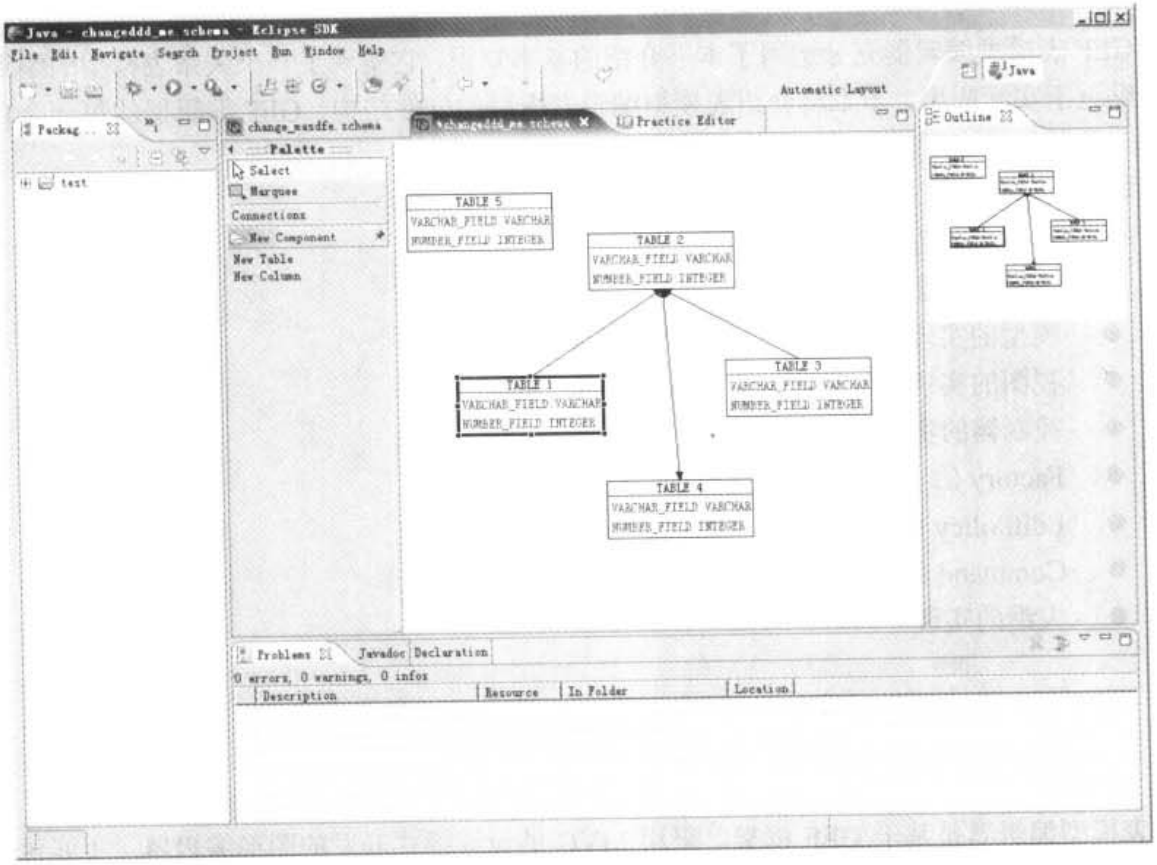


图 21-1 表模型编辑器

21.2 编辑器的实现

本节将通过一个完整的实例，介绍如何实现一个 GEF 编辑器的实现类，具体介绍如下。

21.2.1 编辑器的功能

GEF 的程序都要实现编辑器，编辑器负责 GEF 框架的初始化、模型持久化、注册监听器和获得选项板等工作。编辑器继承于 GraphicalEditor 类，本章的表模型编辑器通过继承 GraphicalEditorWithFlyoutPalette 类实现选项板可伸缩的编辑器。

编辑器就是在编辑器插件中要实现的插件响应类，在开发中，编辑器主要完成如下几个功能。

（1）初始化 GEF 框架。

（2）持久化模型和通过持久化的模型初始化 EditPartViewer（编辑器的 Viewer）。

（3）获得选项板实例，并定制选项板（Palette）。

（4）获得大纲实例。

（5）为 EditPartViewer、PaletteViewer（选项板的 Viewer）和 CommandStack 等注册监听器。

21.2.2　编辑器的实现

编辑器类代表编辑器扩展点的响应类，它能够通过编辑器扩展点中指定扩展名打开相应的文件作为编辑器的输入，打开编辑器及为编辑器添加相应的上下文菜单（Undo、Redo 等）。表模型编辑器如例程 21-1 所示。

例程 21-1　SchemaDiagramEditor.java

```
package com.free.gef.editor;
...//import 描述
public class SchemaDiagramEditor extends GraphicalEditorWithFlyoutPalette
        implements
            CommandStackListener,
            ISelectionListener
{

    //编辑器的底层模型
    private Schema schema;
    private PropertySheetPage undoablePropertySheetPage;
    //编辑器的Viewer
    private GraphicalViewer graphicalViewer;
    private List editPartActionIDs = new ArrayList();
    private List stackActionIDs = new ArrayList();
    private List editorActionIDs = new ArrayList();

    //编辑器大纲
    private OverviewOutlinePage overviewOutlinePage;

    private ActionRegistry actionRegistry;

    private DefaultEditDomain editDomain;

    //编辑器是否被修改过
    private boolean isDirty;

    public SchemaDiagramEditor()
    {
        editDomain = new DefaultEditDomain(this);
        //设置编辑器的Domain
        setEditDomain(editDomain);
    }

    //初始化编辑器
    public void init(IEditorSite site, IEditorInput input) throws PartInitException
```

```
{
    //设置编辑器的site
    setSite(site);
    //设置编辑器的输入
    setInput(input);

    //添加命令堆栈的监听器
    getCommandStack().addCommandStackListener(this);

    //添加选择视图的监听器
    getSite().getWorkbenchWindow().getSelectionService().
        addSelectionListener(this);

    //新建相应的Action
    createActions();

}

//选择相应的视图后响应的方法
public void selectionChanged(IWorkbenchPart part, ISelection selection)
{
    updateActions(editPartActionIDs);
}

//命令堆栈改变后响应方法
public void commandStackChanged(EventObject event)
{
    updateActions(stackActionIDs);
    setDirty(getCommandStack().isDirty());
}

public GraphicalViewer getGraphicalViewer()
{
    return graphicalViewer;
}

public void dispose()
{
    getCommandStack().removeCommandStackListener(this);
    getSite().getWorkbenchWindow().getSelectionService().
        removeSelectionListener(this);
    getActionRegistry().dispose();
    super.dispose();
}

public Object getAdapter(Class adapter)
{
    if (adapter == GraphicalViewer.class || adapter == EditPartViewer.class)
        return getGraphicalViewer();
    else if (adapter == CommandStack.class)
        return getCommandStack();
```

```
    else if (adapter == EditDomain.class)
        return getEditDomain();
    else if (adapter == ActionRegistry.class)
        return getActionRegistry();
    else if (adapter == IPropertySheetPage.class)
        return getPropertySheetPage();
    else if (adapter == IContentOutlinePage.class)
        return getOverviewOutlinePage();

    return super.getAdapter(adapter);
}

public void doSave(IProgressMonitor monitor)
{
    try
    {
        //持久化当前编辑器
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream objectOut = new ObjectOutputStream(out);
        objectOut.writeObject(schema);
        objectOut.close();
        IFile file = ((IFileEditorInput) getEditorInput()).getFile();
        file.setContents(new ByteArrayInputStream(out.toByteArray()),
            true, false, monitor);
        out.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    getCommandStack().markSaveLocation();
}

public void doSaveAs()
{
    throw new UnsupportedOperationException();
}

public boolean isSaveAsAllowed()
{
    return false;
}

public boolean isDirty()
{
    return isDirty;
}

public CommandStack getCommandStack()
{
    return getEditDomain().getCommandStack();
```

```
}

public Schema getSchema()
{
    return schema;
}

protected void setInput(IEditorInput input)
{
    super.setInput(input);

    IFile file = ((IFileEditorInput) input).getFile();
    try
    {
        //用文件的输入初始化编辑器
        setPartName(file.getName());
        InputStream is = file.getContents(true);
        ObjectInputStream ois = new ObjectInputStream(is);
        schema = (Schema) ois.readObject();
        ois.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
        schema = getContent();
    }
}

protected PaletteViewerProvider createPaletteViewerProvider()
{
    return new SchemaPaletteViewerProvider(editDomain);
}


protected void createGraphicalViewer(Composite parent)
{

    IEditorSite editorSite = getEditorSite();
    GraphicalViewer viewer = new GraphicalViewerCreator(editorSite).
        createViewer(parent);

    GraphicalViewerKeyHandler graphicalViewerKeyHandler
        = new GraphicalViewerKeyHandler(viewer);
    KeyHandler parentKeyHandler = graphicalViewerKeyHandler.setParent
        (getCommonKeyHandler());
    viewer.setKeyHandler(parentKeyHandler);

    getEditDomain().addViewer(viewer);

    getSite().setSelectionProvider(viewer);
```

```
    //把schema作为编辑器Viewer的输入
    viewer.setContents(schema);

    ContextMenuProvider provider = new SchemaContextMenuProvider(viewer,
        getActionRegistry());
    //设置编辑器的上下文菜单
    viewer.setContextMenu(provider);
    getSite().registerContextMenu("com.realpersist.gef.editor.contextmenu",
     provider, viewer);

    this.graphicalViewer = viewer;

}

protected KeyHandler getCommonKeyHandler()
{

    KeyHandler sharedKeyHandler = new KeyHandler();
    sharedKeyHandler.put(KeyStroke.getPressed(SWT.DEL, 127, 0),
    getActionRegistry().getAction(
            GEFActionConstants.DELETE));
    sharedKeyHandler.put(KeyStroke.getPressed(SWT.F2, 0),
            getActionRegistry().getAction(
            GEFActionConstants.DIRECT_EDIT));

    return sharedKeyHandler;
}

protected void setDirty(boolean dirty)
{
    if (isDirty != dirty)
    {
        isDirty = dirty;
        //通知Eclipse，当前编辑器已经被修改
        firePropertyChange(IEditorPart.PROP_DIRTY);
    }
}

protected void createActions()
{
    //添加Action
    addStackAction(new UndoAction(this));
    addStackAction(new RedoAction(this));
    addEditPartAction(new DeleteAction((IWorkbenchPart) this));
    addEditorAction(new SaveAction(this));
    addEditorAction(new PrintAction(this));
}

protected void addEditPartAction(SelectionAction action)
{
    getActionRegistry().registerAction(action);
```

```
        editPartActionIDs.add(action.getId());
    }

    protected void addStackAction(StackAction action)
    {
        getActionRegistry().registerAction(action);
        stackActionIDs.add(action.getId());
    }

    protected void addEditorAction(WorkbenchPartAction action)
    {
        getActionRegistry().registerAction(action);
        editorActionIDs.add(action.getId());
    }

    protected void addAction(IAction action)
    {
        getActionRegistry().registerAction(action);
    }

    protected void updateActions(List actionIds)
    {
        for (Iterator ids = actionIds.iterator(); ids.hasNext();)
        {
            IAction action = getActionRegistry().getAction(ids.next());
            if (null != action && action instanceof UpdateAction)
                ((UpdateAction) action).update();

        }
    }

    protected ActionRegistry getActionRegistry()
    {
        if (actionRegistry == null)
            actionRegistry = new ActionRegistry();

        return actionRegistry;
    }

    //得到编辑器的大纲
    protected OverviewOutlinePage getOverviewOutlinePage()
    {
        if (null == overviewOutlinePage && null != getGraphicalViewer())
        {
            RootEditPart rootEditPart = getGraphicalViewer().getRootEditPart();
            if (rootEditPart instanceof ScalableFreeformRootEditPart)
            {
                overviewOutlinePage = new OverviewOutlinePage
                    ((ScalableFreeformRootEditPart) rootEditPart);
            }
        }
```

```
        return overviewOutlinePage;
    }

    protected PropertySheetPage getPropertySheetPage()
    {
        if (null == undoablePropertySheetPage)
        {
            undoablePropertySheetPage = new PropertySheetPage();
            undoablePropertySheetPage.
                setRootEntry(GEFPlugin.createUndoablePropertySheetEntry
                (getCommandStack()));
        }

        return undoablePropertySheetPage;
    }

    protected void firePropertyChange(int propertyId)
    {
        //通知编辑器已经被修改
        super.firePropertyChange(propertyId);
        updateActions(editorActionIDs);
    }

    //返回选项板的参数
    protected FlyoutPreferences getPalettePreferences()
    {
        return new PaletteFlyoutPreferences();
    }

    //得到选项板
    protected PaletteRoot getPaletteRoot()
    {
        return new PaletteViewerCreator().createPaletteRoot();
    }

    private Schema getContent()
    {
        return new ContentCreator().getContent();
    }

    private static ImageDescriptor create(String iconPath, String name)
    {
        return AbstractUIPlugin.imageDescriptorFromPlugin
         ("com.realpersist.gef.schemaeditor", iconPath + name);
    }

}
```

上例为一个完整编辑器的实现，它完成了 GEF 框架初始化、获取选项板和适配大纲等常用编辑器所完成的功能。

21.3 模型的实现

在 GEF 的模型中，要实现一种事件的通知机制，把模型相关的控制器注册为模型的监听器。当模型的属性被修改后，会触发相应的事件，监听器监听到事件后，会根据事件做出相应的处理。

一般来说模型会实现一个基类用来封装事件机制的方法。表模型编辑器中模型的继承关系如图 21-2 所示。

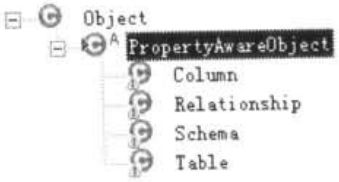


图 21-2 模型的继承关系

PropertyAwareObject 类为模型的基类，表模型编辑器中的模型有 Column、RelationShip、Schema 和 Table。

21.3.1 PropertyAwareObject 类

PropertyAwareObject 类为模型的基类，它主要负责封装所有模型都要用到的事件机制，代码如例程 21-2 所示。

例程 21-2 PropertyAwareObject.java

```
package com.free.gef.model;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public abstract class PropertyAwareObject extends Object implements Serializable
{
    //标识
    public static final String CHILD = "CHILD";
    public static final String REORDER = "REORDER";
    public static final String BOUNDS = "BOUNDS";
    public static final String INPUT = "INPUT";
    public static final String OUTPUT = "OUTPUT";
    public static final String NAME = "NAME";
    public static final String LAYOUT = "LAYOUT";

    protected transient PropertyChangeSupport listeners = new PropertyChangeSupport(this);
```

```
    protected PropertyAwareObject()
    {
    }

    //添加属性改变的监听器
    public void addPropertyChangeListener(PropertyChangeListener l)
    {
        listeners.addPropertyChangeListener(l);
    }

    //删除属性改变的监听器
    public void removePropertyChangeListener(PropertyChangeListener l)
    {
        listeners.removePropertyChangeListener(l);
    }

    //通知属性改变
    protected void firePropertyChange(String prop, Object old, Object newValue)
    {
        listeners.firePropertyChange(prop, old, newValue);
    }

    private void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException
    {
        in.defaultReadObject();
        listeners = new PropertyChangeSupport(this);
    }

}
```

其中，PropertyChangeSupport类型的变量用来触发事件，通知模型的监听器。

21.3.2 Table类

Table为表模型，通过Table模型要能够找到其父模型(Schema)及所有的子模型(Column)，另外Table模型要能保存模型作为主键和外键的所有关系，代码如例程21-3所示。

例程21-3 Table.java

```
package com.free.gef.model;

import java.util.ArrayList;
import java.util.List;
import org.eclipse.draw2d.geometry.Rectangle;

public class Table extends PropertyAwareObject
{

    //表的父模型
    private Schema schema;
```

```
//表名
private String name;
//表中包括的子列，即子模型列表
private ArrayList columns = new ArrayList();
//表的位置和大小
private Rectangle bounds;

//表作为输入（主键）的关系列表
private ArrayList primaryKeyRelationships = new ArrayList();
//表作为输出（外键）的关系列表
private ArrayList foreignKeyRelationships = new ArrayList();

public Table()
{
    super();
}

public Table(String name, Schema schema)
{
    super();
    if (name == null)
        throw new NullPointerException("Name cannot be null");
    if (schema == null)
        throw new NullPointerException("Schema cannot be null");
    this.name = name;
    this.schema = schema;
}

//添加列
public void addColumn(Column column)
{
    if (columns.contains(column))
    {
        throw new IllegalArgumentException("Column already present");
    }
    columns.add(column);
    //触发添加列的事件
    firePropertyChange(CHILD, null, column);
}

public void addColumn(Column column, int index)
{
    if (columns.contains(column))
    {
        throw new IllegalArgumentException("Column already present");
    }
    columns.add(index, column);
    firePropertyChange(CHILD, null, column);
}

//删除列
```

```
public void removeColumn(Column column)
{
    columns.remove(column);
    //触发删除列的事件
    firePropertyChange(CHILD, column, null);
}

public void switchColumn(Column column, int index)
{
    columns.remove(column);
    columns.add(index, column);
    firePropertyChange(REORDER, this, column);
}

public void setName(String name)
{
    this.name = name;
}

public void setSchema(Schema schema)
{
    this.schema = schema;
}

public void setBounds(Rectangle bounds)
{
    this.bounds = bounds;
}

public void addForeignKeyRelationship(Relationship table)
{
    //添加外键关系
    foreignKeyRelationships.add(table);
    firePropertyChange(OUTPUT', null, table);
}

public void addPrimaryKeyRelationship(Relationship table)
{
    //添加主键关系
    primaryKeyRelationships.add(table);
    firePropertyChange(INPUT, null, table);
}

public void removeForeignKeyRelationship(Relationship table)
{
    //删除外键关系
    foreignKeyRelationships.remove(table);
    firePropertyChange(OUTPUT, table, null);
}

public void removePrimaryKeyRelationship(Relationship table)
```

```
{
    //删除主键关系
    primaryKeyRelationships.remove(table);
    firePropertyChange(INPUT, table, null);
}

public void modifyName(String name)
{
    String oldName = this.name;
    if (!name.equals(oldName))
    {
        this.name = name;
        //触发名字改变的事件
        firePropertyChange(NAME, null, name);
    }
}

public void modifyBounds(Rectangle bounds)
{
    Rectangle oldBounds = this.bounds;
    if (!bounds.equals(oldBounds))
    {
        this.bounds = bounds;

        firePropertyChange(BOUNDS, null, bounds);
    }
}

public String getName()
{
    return name;
}

public List getColumns()
{
    return columns;
}

//获得所有的输出关系的模型
public List getForeignKeyRelationships()
{
    return foreignKeyRelationships;
}

//获得所有的输入关系的模型
public List getPrimaryKeyRelationships()
{
    return primaryKeyRelationships;
}

public Schema getSchema()
```

```
    {
        return schema;
    }

    public Rectangle getBounds()
    {
        return bounds;
    }

    public String toString()
    {
        return name;
    }

    public int hashCode()
    {
        if (schema == null || name == null)
            return super.hashCode();
        String schemaName = schema.getName();
        return schemaName.hashCode() + name.hashCode();
    }

    public boolean equals(Object o)
    {
        if (o == null || !(o instanceof Table))
            return false;
        Table t = (Table) o;

        if (schema.getName().equals(t.getSchema().getName()) && name.equals
           (t.getName()))
        {
            return true;
        }
        else
            return false;
    }
}
```

Table模型可以和其他的Table模型建立关系。它维护了两个列表primaryKeyRelationships和foreignKeyRelationships，其中primaryKeyRelationships保存的对象为表模型作为主键的连线模型，foreignKeyRelationships保存的对象为表模型作为外键的连线模型。

另外，Table模型可以包含其他的列模型，它包含了columns列表，columns列表中的对象为列模型。

21.3.3 Column类

Column模型非常简单，它只包含了列名和列的类型，代码如例程21-4所示。

例程 21-4 Column.java

```
package com.free.gef.model;

public class Column extends PropertyAwareObject
{
    //列的类型
    public static String VARCHAR = "VARCHAR";
    public static String INTEGER = "INTEGER";
    public static String DATE = "DATE";

    //列属性
    private String name;
    private String type;

    public Column()
    {
        super();
    }

    public Column(String name, ColumnType type)
    {
        super();
        this.name = name;
        this.type = type.getType();
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        String oldName = this.name;
        this.name = name;
    }

    public String getType()
    {
        return type;
    }

    public void setType(String type)
    {
        String oldType = this.type;
        this.type = type;
        //触发类型改变的事件
        firePropertyChange(NAME, null, type);
    }
```

```
    //获得列的显示文本
    public String getLabelText()
    {
        String labelText = getName() + ":" + getType();
        return labelText;
    }

}
```

21.3.4　Schema 类

Schema 模型是所有模型直接或间接的容器，它包含了所有的表模型。另外本例中，Schema 模型还包含了编辑器的布局方式。实现代码如例程 21-5 所示。

例程 21-5　Schema.java

```
package com.free.gef.model;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Schema extends PropertyAwareObject
{

    //Schema 的名字
    private String name;
    //Schema 中的子表
    private ArrayList tables = new ArrayList();
    private Map tablesMap = new HashMap();
    //是否设定自定义的布局管理器
    private boolean layoutManualDesired = true;
    private boolean layoutManualAllowed = false;

    public Schema(String name)
    {
        super();
        if (name == null)
            throw new NullPointerException("Name cannot be null");
        this.name = name;
    }

    //添加子表，并触发添加了表的事件
    public void addTable(Table table)
    {
        tables.add(table);
        tablesMap.put(table.getName(), table);
        firePropertyChange(CHILD, null, table);
    }
```

```
public void addTable(Table table, int i)
{
    tables.add(i, table);
    tablesMap.put(table.getName(), table);
    firePropertyChange(CHILD, null, table);
}

//删除了表
public void removeTable(Table table)
{
    tables.remove(table);
    tablesMap.remove(table.getName());
    firePropertyChange(CHILD, table, null);
}

public Table getTable(String name)
{
    return (Table) tablesMap.get(name);
}

public List getTables()
{
    return tables;
}

public String getName()
{
    return name;
}

public void setLayoutManualAllowed(boolean layoutManualAllowed)
{
    this.layoutManualAllowed = layoutManualAllowed;
}

public boolean isLayoutManualDesired()
{
    return layoutManualDesired;
}

public void setLayoutManualDesired(boolean layoutManualDesired)
{
    this.layoutManualDesired = layoutManualDesired;
    //通知布局改变
    firePropertyChange(LAYOUT, null, new Boolean(layoutManualDesired));
}

public boolean isLayoutManualAllowed()
{
    return layoutManualAllowed;
}
```

```
}
```

21.3.5 Relationship 类

Relationship 模型是表模型和表模型之间的关系模型，在视图的层面上，Relationship 模型代表一条连线。

Relationship 模型包含了两个表模型属性，一个为 primaryKeyTable，表示主键表模型，另一个为 foreignKeyTable，表示外键表模型，代码如例程 21-6 所示。

例程 21-6 Relationship.java

```
package com.free.gef.model;

public class Relationship extends PropertyAwareObject
{
    //关系（连线）相关联主键的表
    private Table primaryKeyTable;
    //关系（连线）相关联外键的表
    private Table foreignKeyTable;
    private Column foreignKeyColumn;

    public Relationship(Table foreignTable, Table primaryTable)
    {
        super();
        //设置关系的主外键
        this.primaryKeyTable = primaryTable;
        this.foreignKeyTable = foreignTable;
        this.primaryKeyTable.addPrimaryKeyRelationship(this);
        this.foreignKeyTable.addForeignKeyRelationship(this);
    }

    public Column getForeignKeyColumn()
    {
        return foreignKeyColumn;
    }

    public Table getForeignKeyTable()
    {
        return foreignKeyTable;
    }

    public Table getPrimaryKeyTable()
    {
        return primaryKeyTable;
    }

    public void setPrimaryKeyTable(Table targetPrimaryKey)
    {
        this.primaryKeyTable = targetPrimaryKey;
```

```
    }

    public void setForeignKeyTable(Table sourceForeignKey)
    {
        this.foreignKeyTable = sourceForeignKey;
    }
}
```

模型的定义都很简单，它定义了要持久化的数据，任何可持久化数据都可以看做模型。

21.4 视图的实现

在 GEF 中，视图是通过 Draw2d 实现的，如果要表现模型，控制器都会为模型创建对应的视图。

视图是可以复用的，例如，如果有不同的列模型，分别表示整型、日期型和字符型等模型，它们的区别在于显示图标和显示文本，则控制器在创建视图时可以用同一视图类，设置不同的显示图标和显示文本即可。

本例中视图可以分为 TableFigure（表视图）、SchemaFigure（编辑器视图）、ColumnsFigure（列视图）和 PolylineConnection（连线视图），其中 PolylineConnection 为 Draw2d 自带的连线视图。

21.4.1 TableFigure 类

TableFigure 视图代表与 Table 模型对应的图形，代码如例程 21-7 所示。

例程 21-7 TableFigure.java

```
package com.free.gef.figures;

import java.util.List;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.LineBorder;
import org.eclipse.draw2d.ToolbarLayout;
import org.eclipse.swt.graphics.Color;

public class TableFigure extends Figure
{
    //表颜色
    public static Color tableColor = new Color(null, 255, 255, 206);
    //表中包含的列视图
    private ColumnsFigure columnsFigure = new ColumnsFigure();
    //表的可编辑标签
    private EditableLabel nameLabel;

    public TableFigure(EditableLabel name)
```

```
{
    this(name, null);
}

public TableFigure(EditableLabel name, List colums)
{
    //给表的显示标签赋值
    nameLabel = name;
    //设置表中视图的布局方式为 ToolbarLayout
    ToolbarLayout layout = new ToolbarLayout();
    //设置表中的子视图为垂直排列的方式
    layout.setVertical(true);
    layout.setStretchMinorAxis(true);
    setLayoutManager(layout);
    //设置图形的边框
    setBorder(new LineBorder(ColorConstants.black, 1));
    //设置图形的背景色
    setBackgroundColor(tableColor);
    //设置图形的前景色
    setForegroundColor(ColorConstants.black);
    //设置图形是否透明
    setOpaque(true);

    //设置表标签的前景色
    name.setForegroundColor(ColorConstants.black);
    //把标签添加到表图形
    add(name);
    //把列图形加入到表图形中
    add(columnsFigure);

}

//此方法在 TablePart 中调用，当 TablePart 被选中时，加宽表的边框
public void setSelected(boolean isSelected)
{
    LineBorder lineBorder = (LineBorder) getBorder();
    if (isSelected)
    {
        //如果表被选中，设置表的边框为 2
        lineBorder.setWidth(2);
    }
    else
    {
        lineBorder.setWidth(1);
    }
}


public EditableLabel getNameLabel()
{
    return nameLabel;
```

```
    }

    public ColumnsFigure getColumnsFigure()
    {
        return columnsFigure;
    }
}
```

TableFigure 中设置了图形的边框、显示文字及显示的样式等。TableFigure 显示效果如图 21-3 所示。

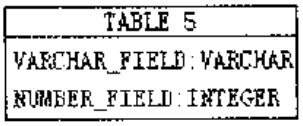


图 21-3 TableFigure 显示效果

21.4.2 SchemaFigure 类

SchemaFigure 代表了底层 Schema 模型对应的视图，它继承于 FreeformLayer（表示可能从四个方向扩展的层），设置图形为透明，代码如例程 21-8 所示。

例程 21-8 SchemaFigure.java

```
package com.free.gef.figures;

import org.eclipse.draw2d.FreeformLayer;

public class SchemaFigure extends FreeformLayer
{

    public SchemaFigure()
    {
        //设置编辑器视图为透明
        setOpaque(true);
    }
}
```

21.4.3 ColumnsFigure 类

ColumnsFigure 是和 Column 模型对应的图形，它表示 TableFigure 图形中一个列的子图形。ColumnsFigure 通过图形中的标签（Label）显示列对应的文本，另外 ColumnsFigure 中实现了边框类，代码如例程 21-9 所示。

例程 21-9 ColumnsFigure.java

```
package com.free.gef.figures;

import org.eclipse.draw2d.AbstractBorder;
```

```
import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.FlowLayout;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.geometry.Insets;

public class ColumnsFigure extends Figure
{
    public ColumnsFigure()
    {
        FlowLayout layout = new FlowLayout();
        //设置对齐方式
        layout.setMinorAlignment(FlowLayout.ALIGN_LEFTTOP);
        layout.setStretchMinorAxis(false);
        layout.setHorizontal(false);
        //设置布局
        setLayoutManager(layout);
        //设置边框
        setBorder(new ColumnFigureBorder());
        //设置背景色
        setBackgroundColor(ColorConstants.tooltipBackground);
        //设置前景色
        setForegroundColor(ColorConstants.blue);
        //设置是否透明
        setOpaque(true);
    }

    //实现边框类
    class ColumnFigureBorder extends AbstractBorder
    {
        public Insets getInsets(IFigure figure)
        {
            return new Insets(5, 3, 3, 1);
        }

        public void paint(IFigure figure, Graphics graphics, Insets insets)
        {
            //重画表的边框
           graphics.drawLine(getPaintRectangle(figure, insets).getTopLeft(),
             tempRect.getTopRight());
        }
    }
}
```

21.5 控制器的实现

控制器是 GEF 中最复杂的一部分，它是模型与视图之间的桥梁，也是整个 GEF 的核心。它不仅要监听模型的变化，当用户编辑视图时，还要把编辑结果反映到模型上。

控制器由一组 EditPart 对象共同组成，每一个模型对象都对应一个 EditPart 对象。在表模型编辑器中，模型有 Table、Column、Schema 和 RelationShip，对应的编辑器有 TablePart、ColumnPart、SchemaDiagramPart 和 RelationshipPart。

控制器生命周期都相同，下面将介绍 TablePart 和 RelationshipPart 的实现，用户可以在本书的源程序中找到其他 EditPart 的实现。

21.5.1 TablePart 的实现

TablePart 是 Table 模型的控制器，它继承于 PropertyAwarePart 类，实现了 NodeEditPart 接口。NodeEditPart 主要提供模型连线的能力。PropertyAwarePart 为 TablePart、ColumnPart 和 SchemaDiagramPart 的基类，它实现将控制器注册为模型监听器，监听模型触发的事件和处理模型连线改变的事件，代码如例程 21-10 所示。

例程 21-10 PropertyAwarePart.java

```
package com.free.gef.part;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.Iterator;
import java.util.List;

import org.eclipse.draw2d.IFigure;
import org.eclipse.gef.ConnectionEditPart;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

import com.free.gef.model.PropertyAwareObject;

public abstract class PropertyAwarePart extends AbstractGraphicalEditPart
implements PropertyChangeListener
{

    public void activate()
    {
        super.activate();
        //注册控制器为模型监听器
        PropertyAwareObject propertyAwareObject = (PropertyAwareObject) getModel();
        propertyAwareObject.addPropertyChangeListener(this);
    }

    public void deactivate()
    {
        super.deactivate();
        //删除模型监听器
        PropertyAwareObject propertyAwareObject = (PropertyAwareObject) getModel();
        propertyAwareObject.removePropertyChangeListener(this);
    }
```

```
public void propertyChange(PropertyChangeEvent evt)
{
    //处理控制器监听事件
    String property = evt.getPropertyName();

    if (PropertyAwareObject.CHILD.equals(property))
    {
        handleChildChange(evt);
    }
    if (PropertyAwareObject.REORDER.equals(property))
    {
        handleReorderChange(evt);
    }
    else if (PropertyAwareObject.OUTPUT.equals(property))
    {
        handleOutputChange(evt);
    }
    else if (PropertyAwareObject.INPUT.equals(property))
    {
        handleInputChange(evt);
    }
    else if (PropertyAwareObject.NAME.equals(property))
    {
        commitNameChange(evt);
    }
    else if (PropertyAwareObject.BOUNDS.equals(property))
    {
        handleBoundsChange(evt);
    }
    else if (PropertyAwareObject.LAYOUT.equals(property))
    {
        handleLayoutChange(evt);
    }

    if (PropertyAwareObject.NAME.equals(property))
    {
        GraphicalEditPart graphicalEditPart = (GraphicalEditPart)
         (getViewer().getContents());
        IFigure partFigure = graphicalEditPart.getFigure();
        partFigure.getUpdateManager().performUpdate();
    }
}

private void handleInputChange(PropertyChangeEvent evt)
{

    //处理输入关系改变的请求
    Object newValue = evt.getNewValue();
    Object oldValue = evt.getOldValue();
```

```
        if (!((oldValue != null) ^ (newValue != null)))
        {
            throw new IllegalStateException("Exactly one of old or new values
                    must be non-null for INPUT event");
        }

        if (newValue != null)
        {
            ConnectionEditPart editPart = createOrFindConnection(newValue);
            int modelIndex = getModelTargetConnections().indexOf(newValue);
            addTargetConnection(editPart, modelIndex);
        }
        else
        {
            List children = getTargetConnections();

            ConnectionEditPart partToRemove = null;
            for (Iterator iter = children.iterator(); iter.hasNext();)
            {
                ConnectionEditPart part = (ConnectionEditPart) iter.next();
                if (part.getModel() == oldValue)
                {
                    partToRemove = part;
                    break;
                }
            }

            if (partToRemove != null)
                removeTargetConnection(partToRemove);
        }

        getContentPane().revalidate();
    }

    private void handleOutputChange(PropertyChangeEvent evt)
    {
        //处理输出关系改变的请求
        Object newValue = evt.getNewValue();
        Object oldValue = evt.getOldValue();

        if (!((oldValue != null) ^ (newValue != null)))
        {
            throw new IllegalStateException(
                "Exactly one of old or new values must be non-null for INPUT event");
        }

        if (newValue != null)
        {
            //添加新的连线
            ConnectionEditPart editPart = createOrFindConnection(newValue);
            int modelIndex = getModelSourceConnections().indexOf(newValue);
```

```
            addSourceConnection(editPart, modelIndex);

        }
        else
        {

            //删除连线
            List children = getSourceConnections();

            ConnectionEditPart partToRemove = null;
            for (Iterator iter = children.iterator(); iter.hasNext();)
            {
                ConnectionEditPart part = (ConnectionEditPart) iter.next();
                if (part.getModel() == oldValue)
                {
                    partToRemove = part;
                    break;
                }
            }

            if (partToRemove != null)
                removeSourceConnection(partToRemove);
        }
        getContentPane().revalidate();
    }

    protected void handleChildChange(PropertyChangeEvent evt)
    {
        //处理子节点改变的请求
        Object newValue = evt.getNewValue();
        Object oldValue = evt.getOldValue();

        if (!((oldValue != null) ^ (newValue != null)))
        {
            throw new IllegalStateException(
                "Exactly one of old or new values must be non-null for CHILD event");
        }

        if (newValue != null)
        {
            //添加新子模型
            EditPart editPart = createChild(newValue);
            int modelIndex = getModelChildren().indexOf(newValue);
            addChild(editPart, modelIndex);

        }
        else
        {
            List children = getChildren();

            EditPart partToRemove = null;
```

```
                for (Iterator iter = children.iterator(); iter.hasNext();)
                {
                    EditPart part = (EditPart) iter.next();
                    if (part.getModel() == oldValue)
                    {
                        partToRemove = part;
                        break;
                    }
                }

                if (partToRemove != null)
                    removeChild(partToRemove);
            }
        }

        protected void handleReorderChange(PropertyChangeEvent evt)
        {
            //刷新视图
            refreshChildren();
            refreshVisuals();
        }

        //在子类中实现相应的方法
        protected void handleLayoutChange(PropertyChangeEvent evt)
        {
        }
        protected void handleBoundsChange(PropertyChangeEvent evt)
        {
        }
        protected void commitNameChange(PropertyChangeEvent evt)
        {
        }
    }
```

PropertyAwarePart 类中实现了 PropertyChangeListener 接口，通过 activate 方法把自己注册为了模型的监听器，另外通过 deactivate 把自己从模型监听器中删除，通过 propertyChange 方法处理模型监听器监听到的事件。

另外，PropertyAwarePart 类通过 handleInputChange 和 handleOutputChange 方法处理了模型连线改变的事件。

TablePart 继承于 PropertyAwarePart 类，实现了 NodeEditPart 接口，通过实现 NodeEditPart 接口实现了视图连线的方式。TablePart 的主要功能是维护 Table 模型、设置 Table 模型的编辑策略、建立模型对应的视图和处理直接编辑。TablePart 实现的代码如例程 21-11 所示。

例程 21-11　TablePart.java

```
package com.free.gef.part;

//import...

public class TablePart extends PropertyAwarePart implements NodeEditPart
```

```
{
    protected DirectEditManager manager;
    //当TablePart创建后调用此方法
    public void activate()
    {
        super.activate();
    }

    //当TablePart销毁时调用此方法
    public void deactivate()
    {
        super.deactivate();
    }

    //下面为和模型相关的方法

    public Table getTable()
    {
        return (Table) getModel();
    }

    //返回Table中的所有子模型
    protected List getModelChildren()
    {
        return getTable().getColumns();
    }

    protected List getModelSourceConnections()
    {
        //获得外键的关系列表
        return getTable().getForeignKeyRelationships();
    }

    protected List getModelTargetConnections()
    {
        //获得主键的关系列表
        return getTable().getPrimaryKeyRelationships();
    }

    //安装TablePart的编辑策略
    protected void createEditPolicies()
    {
        //设置编辑策略
        //模型连线的编辑策略
        installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new TableNodeEditPolicy());
        //子模型布局的编辑策略
        installEditPolicy(EditPolicy.LAYOUT_ROLE, new TableLayoutEditPolicy());
        //处理模型创建请求的编辑策略
        installEditPolicy(EditPolicy.CONTAINER_ROLE, new TableContainerEditPolicy());
        //处理模型编辑的编辑策略，用来处理模型删除的操作
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new TableEditPolicy());
```

```
        //处理直接编辑的编辑策略
        installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new TableDirectEditPolicy());

    }

    public void performRequest(Request request)
    {
        //处理直接编辑的请求
        if (request.getType() == RequestConstants.REQ_DIRECT_EDIT)
        {
            if (request instanceof DirectEditRequest && !directEditHitTest
                    (((DirectEditRequest) request).getLocation().getCopy()))
                return;
            //处理直接编辑
            performDirectEdit();
        }
    }

    //是否允许在视图上直接编辑
    private boolean directEditHitTest(Point requestLoc)
    {
        TableFigure figure = (TableFigure) getFigure();
        EditableLabel nameLabel = figure.getNameLabel();
        nameLabel.translateToRelative(requestLoc);
        if (nameLabel.containsPoint(requestLoc))
            return true;
        return false;
    }

    protected void performDirectEdit()
    {
        if (manager == null)
        {
            //获得编辑器的GraphicalViewer
            ValidationEnabledGraphicalViewer viewer =
                    (ValidationEnabledGraphicalViewer) getViewer();
            ValidationMessageHandler handler = viewer.getValidationHandler();

            TableFigure figure = (TableFigure) getFigure();
            EditableLabel nameLabel = figure.getNameLabel();
            //新建直接编辑管理器
            manager = new ExtendedDirectEditManager(this,
                TextCellEditor.class, new LabelCellEditorLocator(nameLabel),
                nameLabel, new TableNameCellEditorValidator(handler));
        }
        //处理直接编辑
        manager.show();
    }

    public void handleNameChange(String value)
    {
```

```
    TableFigure tableFigure = (TableFigure) getFigure();
    EditableLabel label = tableFigure.getNameLabel();
    label.setVisible(false);
    refreshVisuals();
}

public void revertNameChange()
{
    TableFigure tableFigure = (TableFigure) getFigure();
    EditableLabel label = tableFigure.getNameLabel();
    Table table = getTable();
    label.setText(table.getName());
    label.setVisible(true);
    refreshVisuals();
}

public String toString()
{
    return getModel().toString();
}

//处理表名改变
protected void commitNameChange(PropertyChangeEvent evt)
{
    TableFigure tableFigure = (TableFigure) getFigure();
    EditableLabel label = tableFigure.getNameLabel();
    label.setText(getTable().getName());
    label.setVisible(true);
    refreshVisuals();
}

protected void handleBoundsChange(PropertyChangeEvent evt)
{
    TableFigure tableFigure = (TableFigure) getFigure();
    Rectangle constraint = (Rectangle) evt.getNewValue();
    SchemaDiagramPart parent = (SchemaDiagramPart) getParent();
    parent.setLayoutConstraint(this, tableFigure, constraint);
}

protected IFigure createFigure()
{
    //创建TablePart对应的视图
    Table table = getTable();
    EditableLabel label = new EditableLabel(table.getName());
    TableFigure tableFigure = new TableFigure(label);
    return tableFigure;
}

protected void refreshVisuals()
{
    //刷新视图
```

```
        TableFigure tableFigure = (TableFigure) getFigure();
        Point location = tableFigure.getLocation();
        SchemaDiagramPart parent = (SchemaDiagramPart) getParent();
        Rectangle constraint = new Rectangle(location.x, location.y, -1, -1);
        parent.setLayoutConstraint(this, tableFigure, constraint);
    }

    //获得表中子容器的面板
    public IFigure getContentPane()
    {
        TableFigure figure = (TableFigure) getFigure();
        return figure.getColumnsFigure();
    }

    //获得源连线的锚点
    public ConnectionAnchor getSourceConnectionAnchor(ConnectionEditPart connection)
    {
        return new TopAnchor(getFigure());
    }

    public ConnectionAnchor getSourceConnectionAnchor(Request request)
    {
        return new TopAnchor(getFigure());
    }

    //获得目标连线的锚点
    public ConnectionAnchor getTargetConnectionAnchor(ConnectionEditPart connection)
    {
        return new BottomAnchor(getFigure());
    }

    public ConnectionAnchor getTargetConnectionAnchor(Request request)
    {
        return new BottomAnchor(getFigure());
    }

    public void setSelected(int value)
    {
        super.setSelected(value);
        TableFigure tableFigure = (TableFigure) getFigure();
        //如果选择当前的EditPart，设置tableFigure边框的宽度
        if (value != EditPart.SELECTED_NONE)
            tableFigure.setSelected(true);
        else
            tableFigure.setSelected(false);
        tableFigure.repaint();
    }
}
```

TablePart 通过 performRequest 方法响应直接编辑模型的请求（即用户通过两次单击模型对应的视图，可以直接编辑视图，同在 Windows 中单击文件可以重命名文件一样）。

另外 TablePart 设置了几种类型的编辑策略，用来处理模型连线的请求、模型创建的请求和视图直接编辑后更新模型的请求等。

21.5.2 RelationshipPart 的实现

RelationshipPart 继承于 PropertyAwareConnectionPart 类，PropertyAwareConnectionPart 继承于 AbstractConnectionEditPart 类，实现了 PropertyChangeListener 接口。AbstractConnectionEditPart 实现了连线控制器的基本实现。

PropertyAwareConnectionPart 的主要功能是负责监听器的注册和处理事件，代码如例程 21-12 所示。

例程 21-12 PropertyAwareConnectionPart.java

```
package com.free.gef.part;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.editparts.AbstractConnectionEditPart;

import com.free.gef.model.PropertyAwareObject;

public abstract class PropertyAwareConnectionPart
    extends AbstractConnectionEditPart implements PropertyChangeListener
{
    public void activate()
    {
        super.activate();
        //注册模型监听器
        PropertyAwareObject propertyAwareObject = (PropertyAwareObject) getModel();
        propertyAwareObject.addPropertyChangeListener(this);
    }

    public void deactivate()
    {
        super.deactivate();
        PropertyAwareObject propertyAwareObject = (PropertyAwareObject) getModel();
        propertyAwareObject.removePropertyChangeListener(this);
    }

    public void propertyChange(PropertyChangeEvent evt)
    {
        String property = evt.getPropertyName();
        //处理关系改变的事件
        if (PropertyAwareObject.CHILD.equals(property))
            refreshChildren();
        else if (PropertyAwareObject.INPUT.equals(property))
            refreshTargetConnections();
```

```
            else if (PropertyAwareObject.OUTPUT.equals(property))
                refreshSourceConnections();

            ((GraphicalEditPart) (getViewer().getContents())).getFigure().revalidate();
        }

    }
```

RelationshipPart 表示 Relationship 模型对应的控制器，它主要负责设置编辑策略和创建连线视图，代码如例程 21-13 所示。

例程 21-13　RelationshipPart.java

```
package com.free.gef.part;

import org.eclipse.draw2d.BendpointConnectionRouter;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.PolygonDecoration;
import org.eclipse.draw2d.PolylineConnection;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPolicy;
import org.eclipse.gef.editpolicies.ConnectionEndpointEditPolicy;

import com.free.gef.policy.RelationshipEditPolicy;

public class RelationshipPart extends PropertyAwareConnectionPart
{
    public void activate() {
        super.activate();
    }

    public void deactivate() {
        super.deactivate();
    }

    protected void createEditPolicies()
    {
        //安装编辑策略
        //处理连线请求的编辑策略
        installEditPolicy(EditPolicy.CONNECTION_ENDPOINTS_ROLE,
            new ConnectionEndpointEditPolicy());
        //删除连线的编辑策略
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new RelationshipEditPolicy());
    }

    protected IFigure createFigure()
    {
        //创建连线视图
        PolylineConnection conn = (PolylineConnection) super.createFigure();
        conn.setConnectionRouter(new BendpointConnectionRouter());
        conn.setTargetDecoration(new PolygonDecoration());
        return conn;
```

```
    }

    public void setSelected(int value)
    {
        super.setSelected(value);
        //选择或取消选择当前的EditPart，改变连线视图的连线宽度
        if (value != EditPart.SELECTED_NONE)
            ((PolylineConnection) getFigure()).setLineWidth(2);
        else
            ((PolylineConnection) getFigure()).setLineWidth(1);
    }
}
```

21.6 EditPartFactory 和 Palette 的实现

GEF 中通常都有 EditPartFactory 和 Palette 的相应类用于实现模型、控制器和选项板的创建，具体介绍如下。

21.6.1 SchemaEditPartFactory

在 GEF 中，模型和控制器是一一对应的，即添加一个模型后，框架会对应添加一个控制器并把模型和控制器关联起来。

EditPartFactory 是一个控制器工厂的接口，通过实现 createEditPart 方法能够创建模型对应的控制器，当 GEF 框架创建了一个模型后，会调用 createEditPart 方法创建对应的控制器，并把模型和控制器关联起来。本应用中控制器工厂为 SchemaEditPartFactory，它实现了 EditPartFactory 接口，代码如例程 21-14 所示。

例程 21-14 SchemaEditPartFactory.java

```
package com.free.gef.part.factory;

import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPartFactory;

import com.free.gef.model.Column;
import com.free.gef.model.Relationship;
import com.free.gef.model.Schema;
import com.free.gef.model.Table;
import com.free.gef.part.ColumnPart;
import com.free.gef.part.RelationshipPart;
import com.free.gef.part.SchemaDiagramPart;
import com.free.gef.part.TablePart;

public class SchemaEditPartFactory implements EditPartFactory
{
    public EditPart createEditPart(EditPart context, Object model)
    {
```

```
            EditPart part = null;
            //新建模型对应的控制器
            if (model instanceof Schema)
                part = new SchemaDiagramPart();
            else if (model instanceof Table)
                part = new TablePart();
            else if (model instanceof Relationship)
                part = new RelationshipPart();
            else if (model instanceof Column)
                part = new ColumnPart();

            //把模型和控制器关联起来
            part.setModel(model);
            return part;
        }
    }
```

如上例所示，框架每创建一个模型都会创建对应的控制器，并把模型和控制器关联起来。但是如何让 GEF 框架知道用户自定义的控制器工厂呢？这就要求用户在编辑器初始化建立 GraphicalViewer 时，设置控制器工厂，例如“viewer.setEditPartFactory(new SchemaEditPart Factory());”。

21.6.2 PaletteRoot

在 GEF 的编辑器中要求实现 getPaletteRoot 方法，返回编辑器的选项板。本应用中通过 PaletteViewerCreator 的 createPaletteRoot 方法创建编辑器的选项板，代码如例程 21-15 所示。

例程 21-15 PaletteViewerCreator.java

```
package com.free.gef.editor;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.gef.palette.CombinedTemplateCreationEntry;
import org.eclipse.gef.palette.ConnectionCreationToolEntry;
import org.eclipse.gef.palette.MarqueeToolEntry;
import org.eclipse.gef.palette.PaletteDrawer;
import org.eclipse.gef.palette.PaletteEntry;
import org.eclipse.gef.palette.PaletteGroup;
import org.eclipse.gef.palette.PaletteRoot;
import org.eclipse.gef.palette.PaletteSeparator;
import org.eclipse.gef.palette.SelectionToolEntry;
import org.eclipse.gef.palette.ToolEntry;
import org.eclipse.ui.plugin.AbstractUIPlugin;

import com.free.gef.dnd.DataElementFactory;
import com.free.gef.model.Column;
import com.free.gef.model.Table;

public class PaletteViewerCreator
```

```
{
    private PaletteRoot paletteRoot;

    public PaletteRoot createPaletteRoot()
    {
        //创建 PaletteRoot
        paletteRoot = new PaletteRoot();

        //创建默认的选项板分组
        PaletteGroup controls = new PaletteGroup("Controls");
        paletteRoot.add(controls);

        //创建系统自带的 SelectionToolEntry 选择选项板（箭头）
        ToolEntry tool = new SelectionToolEntry();
        controls.add(tool);

        //设置 tool 为默认的选择项
        paletteRoot.setDefaultEntry(tool);

        //添加 MarqueeToolEntry 选项板项
        controls.add(new MarqueeToolEntry());

        //在分组中添加分隔项
        PaletteSeparator separator = new PaletteSeparator(
                SchemaDiagramPlugin.PLUGIN_ID + ".palette.seperator");
        separator.setUserModificationPermission(
                PaletteEntry.PERMISSION_NO_MODIFICATION);
        controls.add(separator);

        //在分组中添加选择连线模型项
        controls.add(new ConnectionCreationToolEntry(
                "Connections", "Create Connections", null,
                AbstractUIPlugin.imageDescriptorFromPlugin(
                    SchemaDiagramPlugin.PLUGIN_ID,"icons/relationship.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                    SchemaDiagramPlugin.PLUGIN_ID, "icons/relationship.gif")));

        //添加可伸缩的分组
        PaletteDrawer drawer = new PaletteDrawer("New Component",
                AbstractUIPlugin.imageDescriptorFromPlugin(
                SchemaDiagramPlugin.PLUGIN_ID, "icons/connection.gif"));

        List entries = new ArrayList();

        //添加 Table 模型对应选项板的选项
        CombinedTemplateCreationEntry tableEntry =
            new CombinedTemplateCreationEntry("New Table", "Create a new table",
                Table.class, new DataElementFactory(Table.class),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        SchemaDiagramPlugin.PLUGIN_ID, "icons/table.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
```

```
                                SchemaDiagramPlugin.PLUGIN_ID, "icons/table.gif"));

            //添加Column模型对应选项板的选项
            CombinedTemplateCreationEntry columnEntry =
                new CombinedTemplateCreationEntry("New Column", "Add a new column",
                    Column.class, new DataElementFactory(Column.class),
                    AbstractUIPlugin.imageDescriptorFromPlugin(
                            SchemaDiagramPlugin.PLUGIN_ID, "icons/column.gif"),

                    AbstractUIPlugin.imageDescriptorFromPlugin(
                            SchemaDiagramPlugin.PLUGIN_ID, "icons/column.gif"));

            entries.add(tableEntry);
            entries.add(columnEntry);

            //把Table模型和Column模型对应选项添加到可伸缩分组中
            drawer.addAll(entries);

            paletteRoot.add(drawer);

            //返回paletteRoot
            return paletteRoot;

        }

        public PaletteRoot getPaletteRoot()
        {
            return paletteRoot;
        }
    }
```

从例程可以看出，在选项板中添加了 SelectionToolEntry 和 MarqueeToolEntry 两个选项，分别代表选择和选取框选项。

21.7 EditPolicy 的实现

EditPart(控制器)中通过 createEditPolicies 方法添加 EditPolicy(编辑策略)，每种 EditPolicy 负责处理相应的请求。EditPart 会遍历所有的 EditPolicy 处理请求，每种类型的 EditPolicy 关心特定的请求，而忽略其他的请求。

每个 EditPart 都可以设置多个 EditPolicy，通过 EditPolicy，控制器能够处理多种类型的请求，常用的请求有创建模型、删除模型、移动模型和改变模型大小等，读者可以参考 GEF 的联机帮助文档了解所有的请求。本节就 TablePart 中所添加的 EditPolicy 介绍控制器处理请求，其他的 EditPart 中的 EditPolicy，读者可以查找本书的源代码。

21.7.1 TableNodeEditPolicy（模型连线的编辑策略）

在 TablePart 中通过“installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new TableNode-

EditPolicy());”语句设置了 TablePart 的 TableNodeEditPolicy 编辑策略。

TableNodeEditPolicy 继承于 GraphicalNodeEditPolicy 类，GraphicalNodeEditPolicy 类是处理模型连线的编辑策略，在要求模型连线的应用中，一般要处理几种类型的请求，包括创建模型之间的关系（即创建连线）、完成模型的连线、重定位连线输入端的位置和重定位连线输出端的位置，代码如例程 21-16 所示。

例程 21-16　TableNodeEditPolicy.java

```
package com.free.gef.policy;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.GraphicalNodeEditPolicy;
import org.eclipse.gef.requests.CreateConnectionRequest;
import org.eclipse.gef.requests.ReconnectRequest;

import com.free.gef.command.ReconnectForeignKeyCommand;
import com.free.gef.command.ReconnectPrimaryKeyCommand;
import com.free.gef.command.RelationshipCreateCommand;
import com.free.gef.model.Relationship;
import com.free.gef.part.TablePart;

public class TableNodeEditPolicy extends GraphicalNodeEditPolicy
{
    //创建连线
    protected Command getConnectionCreateCommand(CreateConnectionRequest request)
    {
        //创建连线的命令
        RelationshipCreateCommand cmd = new RelationshipCreateCommand();
        TablePart part = (TablePart) getHost();
        //设置外键的表（输出）
        cmd.setForeignTable(part.getTable());
        //把命令放入请求中
        request.setStartCommand(cmd);
        return cmd;
    }

    //完成连线
    protected Command getConnectionCompleteCommand(CreateConnectionRequest request)
    {
        //获得创建连线的命令
        RelationshipCreateCommand cmd = (RelationshipCreateCommand)
            request.getStartCommand();
        TablePart part = (TablePart) request.getTargetEditPart();
        //设置主表（输入）
        cmd.setPrimaryTable(part.getTable());
        return cmd;
    }

    //重定位输出连线
    protected Command getReconnectSourceCommand(ReconnectRequest request)
```

```
        {
            //创建重定位输出连线的命令
            ReconnectForeignKeyCommand cmd = new ReconnectForeignKeyCommand();
            cmd.setRelationship((Relationship)
                    request.getConnectionEditPart().getModel());
            TablePart tablePart = (TablePart) getHost();
            //设置连线输出端的表
            cmd.setSourceForeignKey(tablePart.getTable());
            return cmd;
        }

        //重定位输入连线
        protected Command getReconnectTargetCommand(ReconnectRequest request)
        {
            //创建重定位输入连线的命令
            ReconnectPrimaryKeyCommand cmd = new ReconnectPrimaryKeyCommand();
            cmd.setRelationship((Relationship)
                    request.getConnectionEditPart().getModel());
            TablePart tablePart = (TablePart) getHost();
            //设置连线输入端的表
            cmd.setTargetPrimaryKey(tablePart.getTable());
            return cmd;
        }
    }
```

通过 TableNodeEditPolicy，用户可以创建表和表之间的连线，另外还能够通过拖动重新定位连线输入端和输出端的位置。

21.7.2 TableLayoutEditPolicy（子模型布局的编辑策略）

在 TablePart 中通过“installEditPolicy(EditPolicy.LAYOUT_ROLE, new TableLayoutEditPolicy());”语句设置了 TablePart 的 TableLayoutEditPolicy 编辑策略。

TableLayoutEditPolicy 继承于 FlowLayoutEditPolicy，FlowLayoutEditPolicy 是一种布局型的编辑策略，当把一个列从一列拖到另外一列，或把一列从表的一个位置拖到另一个位置，布局的信息就改变了。TableLayoutEditPolicy 能处理类似的请求，代码如例程 21-17 所示。

例程 21-17 TableLayoutEditPolicy.java

```
package com.free.gef.policy;

import org.eclipse.gef.EditPart;
import org.eclipse.gef.Request;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.FlowLayoutEditPolicy;
import org.eclipse.gef.requests.CreateRequest;

import com.free.gef.command.ColumnMoveCommand;
import com.free.gef.command.ColumnTransferCommand;
import com.free.gef.model.Column;
import com.free.gef.model.Table;
```

```
import com.free.gef.part.ColumnPart;
import com.free.gef.part.TablePart;

public class TableLayoutEditPolicy extends FlowLayoutEditPolicy
{
    //添加子模型
    protected Command createAddCommand(EditPart child, EditPart after)
    {
        if (!(child instanceof ColumnPart))
            return null;
        if (!(after instanceof ColumnPart))
            return null;

        //得到被移动的模型
        Column toMove = (Column) child.getModel();
        Column afterModel = (Column) after.getModel();

        TablePart originalTablePart = (TablePart) child.getParent();
        //被移动模型的父模型
        Table originalTable = (Table) originalTablePart.getModel();
        TablePart newTablePart = (TablePart) after.getParent();
        //被移动到的新模型
        Table newTable = newTablePart.getTable();

        //被移动模型在原来模型中的显示顺序
        int oldIndex = originalTablePart.getChildren().indexOf(child);
        //被移动模型在新模型中的显示顺序
        int newIndex = newTablePart.getChildren().indexOf(after);

        //创建列模型从一个表到另一个表的命令
        ColumnTransferCommand command =
            new ColumnTransferCommand(toMove, afterModel, originalTable,
                                      newTable,oldIndex, newIndex);
        return command;
    }

    protected Command createMoveChildCommand(EditPart child, EditPart after)
    {
        if (after != null)
        {
            //得到被移动的模型
            Column childModel = (Column) child.getModel();
            Column afterModel = (Column) after.getModel();

            Table parentTable = (Table) getHost().getModel();
            //得到模型的原始位置
            int oldIndex = getHost().getChildren().indexOf(child);
            //得到模型的新的位置
            int newIndex = getHost().getChildren().indexOf(after);

            //创建列模型在表中移动的命令
```

```
                ColumnMoveCommand command = new ColumnMoveCommand
                        (childModel, parentTable, oldIndex, newIndex);
                return command;
            }
            return null;
        }

        protected Command getCreateCommand(CreateRequest request)
        {
            return null;
        }

        protected Command getDeleteDependantCommand(Request request)
        {
            return null;
        }
    }
```

在 TableLayoutEditPolicy 中，通过 createAddCommand 处理列模型在表和表之间拖动的请求，通过 createMoveChildCommand 处理列模型在表内部拖动的请求。

21.7.3 TableContainerEditPolicy（处理模型创建请求的编辑策略）

在 TablePart 中通过“installEditPolicy(EditPolicy.CONTAINER_ROLE, new TableContainer EditPolicy());”语句设置了 TablePart 的 TableContainerEditPolicy 编辑策略。

TableContainerEditPolicy 继承于 ContainerEditPolicy，ContainerEditPolicy 可以用来处理容器中添加子模型的请求。TableContainerEditPolicy 实现代码如例程 21-18 所示。

例程 21-18 TableContainerEditPolicy.java

```
package com.free.gef.policy;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.ContainerEditPolicy;
import org.eclipse.gef.requests.CreateRequest;

import com.free.gef.command.ColumnCreateCommand;
import com.free.gef.model.Column;
import com.free.gef.model.Table;
import com.free.gef.part.TablePart;

public class TableContainerEditPolicy extends ContainerEditPolicy
{
    protected Command getCreateCommand(CreateRequest request)
    {
        Object newObject = request.getNewObject();
        //如果表中添加的是列，则添加列模型
        if (!(newObject instanceof Column))
        {
            return null;
```

```
        }

        //得到表模型
        TablePart tablePart = (TablePart) getHost();
        Table table = tablePart.getTable();
        //得到创建的列模型
        Column column = (Column) newObject;
        //创建列模型创建的命令
        ColumnCreateCommand command = new ColumnCreateCommand();
        command.setTable(table);
        command.setColumn(column);
        return command;
    }
}
```

TableContainerEditPolicy 通过 getCreateCommand 处理列模型创建的请求，并通过命令 ColumnCreateCommand 把列模型添加到当前的表模型中。

21.7.4 TableEditPolicy（模型编辑的编辑策略）

在 TablePart 中通过“installEditPolicy(EditPolicy.COMPONENT_ROLE, new TableEditPolicy());”语句设置了 TablePart 的 TableEditPolicy 编辑策略。

TableEditPolicy 继承于 ComponentEditPolicy，当模型被选中后，ComponentEditPolicy 会调用 createDeleteCommand 获得删除模型的命令，当用户执行删除模型的动作（如按【Del】键）时，GEF 框架会执行删除模型的命令。TableEditPolicy 代码如例程 21-19 所示。

例程 21-19 TableEditPolicy.java

```
package com.free.gef.policy;

import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.ComponentEditPolicy;
import org.eclipse.gef.requests.GroupRequest;

import com.free.gef.command.DeleteTableCommand;
import com.free.gef.model.Schema;
import com.free.gef.model.Table;
import com.free.gef.part.TablePart;

public class TableEditPolicy extends ComponentEditPolicy
{

    protected Command createDeleteCommand(GroupRequest request)
    {
        //获得被选择模型的控制器
        TablePart tablePart = (TablePart) getHost();
        //获得模型的大小位置
        Rectangle bounds = tablePart.getFigure().getBounds().getCopy();
        //获得父模型
```

```
        Schema parent = (Schema) (tablePart.getParent().getModel());
        //创建删除模型的命令
        DeleteTableCommand deleteCmd = new DeleteTableCommand();
        deleteCmd.setSchema(parent);
        deleteCmd.setTable((Table) (tablePart.getModel()));
        deleteCmd.setOriginalBounds(bounds);
        return deleteCmd;
    }
    }
```

21.7.5 TableDirectEditPolicy（处理直接编辑的编辑策略）

在 TablePart 中通过“installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new TableDirectEditPolicy());”语句设置了 TablePart 的 TableDirectEditPolicy 编辑策略。

TableDirectEditPolicy 继承于 DirectEditPolicy，当对表进行直接编辑时会处理相应的请求，当对表直接编辑完成后，会通过相应的命令把编辑后的内容保存到表模型中，代码如例程 21-20 所示。

例程 21-20 TableDirectEditPolicy.java

```
package com.free.gef.policy;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.DirectEditPolicy;
import org.eclipse.gef.requests.DirectEditRequest;
import org.eclipse.jface.viewers.CellEditor;

import com.free.gef.command.ChangeTableNameCommand;
import com.free.gef.model.Table;
import com.free.gef.part.TablePart;

public class TableDirectEditPolicy extends DirectEditPolicy
{

    private String oldValue;

    protected Command getDirectEditCommand(DirectEditRequest request)
    {
        //直接编辑后，把编辑后的数据保存在模型中
        ChangeTableNameCommand cmd = new ChangeTableNameCommand();
        Table table = (Table) getHost().getModel();
        cmd.setTable(table);
        cmd.setOldName(table.getName());
        //得到编辑的单元格
        CellEditor cellEditor = request.getCellEditor();
        //设置编辑后的数据到命令中
        cmd.setName((String) cellEditor.getValue());
        return cmd;
    }
```

```
    //显示当前的值
    protected void showCurrentEditValue(DirectEditRequest request)
    {
        String value = (String) request.getCellEditor().getValue();
        TablePart tablePart = (TablePart) getHost();
        tablePart.handleNameChange(value);
    }

    //保存原始值
    protected void storeOldEditValue(DirectEditRequest request)
    {

        CellEditor cellEditor = request.getCellEditor();
        oldValue = (String) cellEditor.getValue();
    }

    protected void revertOldEditValue(DirectEditRequest request)
    {
        CellEditor cellEditor = request.getCellEditor();
        cellEditor.setValue(oldValue);
        TablePart tablePart = (TablePart) getHost();
        tablePart.revertNameChange();
    }
}
```

要想直接编辑后把数据保存到模型中，用 getDirectEditCommand 方法可以完成相应的操作。

21.8 Command 的实现

EditPolicies 通过 Command 处理请求，处理响应请求的方法将返回用户新建的 Command 命令，如果返回的命令不为空的话，GEF 框架会在适当的时间执行命令，从而完成对模型的操作。

GEF 框架维护了一个 CommandStack（命令堆栈），框架会把执行的命令添加到堆栈中。CommandStack 管理命令的 execute（执行）、undo（撤销）和 redo（重做）操作，所以在 GEF 中对模型的任何操作都是可以撤销和重做的，用户要实现的是 Command 的 undo 和 redo 方法。

本节将介绍表模型编辑器 TablePart 的 EditPolicies 中创建的 Command，包括表模型连线命令、模型拖动命令、模型创建命令、模型删除命令和模型直接编辑命令，其他 Command 的实现用户可以参考本书源代码。

21.8.1 模型连线命令

TablePart 添加了 TableNodeEditPolicy 编辑策略，TableNodeEditPolicy 主要负责表模型连线的建立。模型的连线分为创建表和表之间连线的命令（RelationshipCreateCommand），重定位输出连线的命令（ReconnectForeignKeyCommand）和重定位输入连线的命令（Reconnect

PrimaryKeyCommand)。下面将介绍 RelationshipCreateCommand，其他两个 Command 实现方式差不多，读者可参考本书源程序。

RelationshipCreateCommand 创建表和表之间的连线，其中有一个表是主表(primaryTable)，有一个表是子表（foreignTable），另外 RelationshipCreateCommand 新建了一个关系的模型（relationship），代码如例程 21-21 所示。

例程 21-21 RelationshipCreateCommand.java

```
package com.free.gef.command;

import java.util.List;
import org.eclipse.gef.commands.Command;
import com.free.gef.model.Relationship;
import com.free.gef.model.Table;

public class RelationshipCreateCommand extends Command
{
    //连线模型
    protected Relationship relationship;
    //作为外键的表（连线为输出）
    protected Table foreignTable;
    //作为主键的表（连线为输入）
    protected Table primaryTable;

    //是否能执行命令
    public boolean canExecute()
    {
        boolean returnValue = true;
        //如果输入和输出的表是同一个表，则不允许框架执行命令
        if (foreignTable.equals(primaryTable))
        {
            returnValue = false;
        }
        else
        {
            //如果主表为空，则不允许框架执行命令
            if (primaryTable == null)
            {
                return false;
            }
            else
            {
                //如果主表已经有此关系，则不允许框架执行命令
                List relationships = primaryTable.getPrimaryKeyRelationships();
                for (int i = 0; i < relationships.size(); i++)
                {
                    Relationship currentRelationship = (Relationship)
                        relationships.get(i);
                    if (currentRelationship.getForeignKeyTable().
                        equals(foreignTable))
```

```
                {
                    returnValue = false;
                    break;
                }
            }
        }

    }
    return returnValue;
}

//执行命令
public void execute()
{
    //建立表和表之间的关系
    relationship = new Relationship(foreignTable, primaryTable);
}

public Table getForeignTable()
{
    return foreignTable;
}

public Table getPrimaryTable()
{
    return primaryTable;
}

public Relationship getRelationship()
{
    return relationship;
}

//重做命令
public void redo()
{
    //添加主表和子表之间的连线关系
    foreignTable.addForeignKeyRelationship(relationship);
    primaryTable.addPrimaryKeyRelationship(relationship);
}

public void setForeignTable(Table foreignTable)
{
    this.foreignTable = foreignTable;
}

public void setPrimaryTable(Table primaryTable)
{
    this.primaryTable = primaryTable;
}
```

```
    public void setRelationship(Relationship relationship)
    {
        this.relationship = relationship;
    }

    //撤销命令
    public void undo()
    {
        //删除主表和子表之间的连线关系
        foreignTable.removeForeignKeyRelationship(relationship);
        primaryTable.removePrimaryKeyRelationship(relationship);
    }
}
```

TableNodeEditPolicy 中响应请求时会创建 RelationshipCreateCommand 命令，并为命令赋值。RelationshipCreateCommand 中通过 execute 执行命令，GEF 框架会根据 canExecute 方法判断命令是否能被执行，另外通过 undo 方法执行命令的撤销，以及通过 redo 方法执行命令的重做。

21.8.2 模型拖动命令

TableLayoutEditPolicy 中通过 ColumnTransferCommand 命令实现把列模型从一个表移动到另一个表，通过 ColumnMoveCommand 命令实现列模型在表内的移动，两个命令的实现原理类似。下面将介绍 ColumnTransferCommand 的实现，代码如例程 21-22 所示，用户可以参考本书源程序了解 ColumnMoveCommand 的实现。

例程 21-22 ColumnTransferCommand.java

```
package com.free.gef.command;
import org.eclipse.gef.commands.Command;
import com.free.gef.model.Column;
import com.free.gef.model.Table;

public class ColumnTransferCommand extends Command
{

    //被拖动的列模型
    private Column columnToMove;
    private Column columnAfter;
    //从originalTable表拖动模型
    private Table originalTable;
    //拖动模型到newTable表中
    private Table newTable;
    //模型在被拖动表的位置和在新表的位置
    private int oldIndex, newIndex;

    public ColumnTransferCommand(Column columnToMove, Column columnAfter,
            Table originalTable, Table newTable, int oldIndex, int newIndex)
```

```
    {
        super();
        this.columnToMove = columnToMove;
        this.columnAfter = columnAfter;
        this.originalTable = originalTable;
        this.newTable = newTable;
        this.oldIndex = oldIndex;
        this.newIndex = newIndex;
    }

    public boolean canExecute()
    {
        //如果原表中列模型不为空，则可执行命令
        if (originalTable.getColumns().size() > 1)
            return true;
        else
            return false;
    }

    public void execute()
    {
        //在原表中删除了模型
        originalTable.removeColumn(columnToMove);
        //在新表中添加了模型
        newTable.addColumn(columnToMove, newIndex);
    }

    //撤销
    public void undo()
    {
        //在新表中删除模型
        newTable.removeColumn(columnToMove);
        //在原表的原位置添加模型
        originalTable.addColumn(columnToMove, oldIndex);
    }
}
```

ColumnTransferCommand 实现了 canExecute、execute 和 undo 方法，没有实现 redo 方法。如果命令没有实现 redo 方法，当用户选择重做时，将调用父类的 redo 方法，即执行 execute 方法。本例中 redo 则会执行把模型从一个表拖动到另一个表。

21.8.3　模型创建命令

TableContainerEditPolicy 中通过 ColumnCreateCommand 命令把创建子模型添加到表模型中，代码如例程 21-23 所示。

例程 21-23　ColumnCreateCommand.java

```
package com.free.gef.command;
```

```
import org.eclipse.gef.commands.Command;
import com.free.gef.model.Column;
import com.free.gef.model.Table;

public class ColumnCreateCommand extends Command
{
    //新的列模型
    private Column column;
    private Table table;

    public void setColumn(Column column)
    {
        this.column = column;
        //设置列的显示文件
        this.column.setName("COLUMN " + (table.getColumns().size() + 1));
        this.column.setType(Column.VARCHAR);
    }

    public void setTable(Table table)
    {
        this.table = table;
    }

    public void execute()
    {
        //在表中添加列
        table.addColumn(column);
    }

    public void undo()
    {
        //从表中删除列
        table.removeColumn(column);
    }
}
```

当 GEF 创建了一个列模型后，会通过 ColumnCreateCommand 命令的 execute 方法把列添加到表中。

21.8.4 模型删除命令

用户能创建模型，相应地也能删除模型。TableEditPolicy 会通过 createDeleteCommand 方法返回删除模型的命令 DeleteTableCommand。当用户选择表模型后，GEF 框架会通过 createDeleteCommand 方法得到 DeleteTableCommand，但不会立即执行，只有当用户按“Del”键或选择相应的菜单时才会执行 DeleteTableCommand 命令。DeleteTableCommand 代码如例程 21-24 所示。

例程 21-24 DeleteTableCommand.java

```
package com.free.gef.command;
```

```
import java.util.ArrayList;
import java.util.List;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.commands.Command;
import com.free.gef.model.Relationship;
import com.free.gef.model.Schema;
import com.free.gef.model.Table;

public class DeleteTableCommand extends Command
{
    //被选择的表模型
    private Table table;
    //表模型的父模型
    private Schema schema;
    private int index = -1;

    private List foreignKeyRelationships = new ArrayList();
    private List primaryKeyRelationships = new ArrayList();
    private Rectangle bounds;

    private void deleteRelationships(Table t)
    {
        //获得表关联的所有外键的连线
        this.foreignKeyRelationships.addAll(t.getForeignKeyRelationships());

        //删除所有关联的外键的连线
        for (int i = 0; i < foreignKeyRelationships.size(); i++)
        {
            Relationship r = (Relationship) foreignKeyRelationships.get(i);
            r.getPrimaryKeyTable().removePrimaryKeyRelationship(r);
            t.removeForeignKeyRelationship(r);
        }

        this.primaryKeyRelationships.addAll(t.getPrimaryKeyRelationships());
        //删除所有关联的主键的连线
        for (int i = 0; i < primaryKeyRelationships.size(); i++)
        {
            Relationship r = (Relationship) primaryKeyRelationships.get(i);
            r.getForeignKeyTable().removeForeignKeyRelationship(r);
            t.removePrimaryKeyRelationship(r);
        }
    }

    public void execute()
    {
        primExecute();
    }

    protected void primExecute()
    {
```

```
        //删除表关联的连线
        deleteRelationships(table);
        index = schema.getTables().indexOf(table);
        //删除表
        schema.removeTable(table);
    }

    public void redo()
    {
        primExecute();
    }

    private void restoreRelationships()
    {
        //恢复所有外键的连线
        for (int i = 0; i < foreignKeyRelationships.size(); i++)
        {
            Relationship r = (Relationship) foreignKeyRelationships.get(i);
            r.getForeignKeyTable().addForeignKeyRelationship(r);
            r.getPrimaryKeyTable().addPrimaryKeyRelationship(r);
        }
        foreignKeyRelationships.clear();
        //恢复所有主键的连线
        for (int i = 0; i < primaryKeyRelationships.size(); i++)
        {
            Relationship r = (Relationship) primaryKeyRelationships.get(i);
            r.getForeignKeyTable().addForeignKeyRelationship(r);
            r.getPrimaryKeyTable().addPrimaryKeyRelationship(r);
        }
        primaryKeyRelationships.clear();
    }

    public void setTable(Table a)
    {
        table = a;
    }

    public void setSchema(Schema sa)
    {
        schema = sa;
    }

    public void undo()
    {
        //添加表
        schema.addTable(table, index);
        //恢复表的连线
        restoreRelationships();
        table.modifyBounds(bounds);
    }
```

```
    public void setOriginalBounds(Rectangle bounds)
    {
        this.bounds = bounds;
    }
}
```

删除表模型的同时，用户应该删除模型对应的连线，当用户选择撤销时要恢复表模型对应的连线。上例采用 foreignKeyRelationships 和 primaryKeyRelationships 列表保存模型对应的主键和外键连线，当撤销删除时会从 foreignKeyRelationships 和 primaryKeyRelationships 恢复连线。

21.8.5 模型直接编辑命令

TablePart 通过 performRequest 处理用户的直接编辑请求，当用户两次单击模型对应的视图时会显示直接编辑的界面。但是模型不知道任何编辑的信息，当编辑完成后，应该通过命令把编辑后的信息保存在模型中。

TableDirectEditPolicy 通过 ChangeTableNameCommand 来更新模型的数据，代码如例程 21-25 所示。

例程 21-25 ChangeTableNameCommand.java

```
package com.free.gef.command;

import org.eclipse.gef.commands.Command;
import com.free.gef.model.Table;

public class ChangeTableNameCommand extends Command
{
    //表模型
    private Table table;
    //新的表名和原表名
    private String name, oldName;

    public void execute()
    {
        //修改表模型的名字
        table.modifyName(name);
    }

    public boolean canExecute()
    {
        //如果名字不为空，则允许执行命令
        if (name != null)
        {
            return true;
        }
        else
        {
            name = oldName;
```

```
            return false;
        }
    }

    public void setName(String string)
    {
        this.name = string;
    }

    public void setOldName(String string)
    {
        oldName = string;
    }

    public void setTable(Table table)
    {
        this.table = table;
    }

    public void undo()
    {
        //撤销名字的更改
        table.modifyName(oldName);
    }
}
```

21.9 大纲的实现

大纲视图是 Eclipse 自带的视图，它能够显示编辑器中的一些元素信息，例如在 Java 文件的编辑器中，大纲视图显示了类的方法名和属性名，双击大纲中的节点还能定位到文件的相应位置。

在编辑器中可以通过 getAdapter 方法适配一个大纲的实例，从而在编辑器中实现自定义的大纲视图。本章的大纲显示了编辑器的一个缩略图，代码如例程 21-26 所示。

例程 21-26 OverviewOutlinePage.java

```
package com.free.gef.editor;

import org.eclipse.draw2d.LightweightSystem;
import org.eclipse.draw2d.MarginBorder;
import org.eclipse.draw2d.Viewport;
import org.eclipse.draw2d.parts.ScrollableThumbnail;
import org.eclipse.draw2d.parts.Thumbnail;
import org.eclipse.gef.LayerConstants;
import org.eclipse.gef.editparts.ScalableFreeformRootEditPart;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.ISelectionChangedListener;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.swt.SWT;
```

```
import org.eclipse.swt.widgets.Canvas;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.ui.part.Page;
import org.eclipse.ui.views.contentoutline.IContentOutlinePage;

public class OverviewOutlinePage extends Page implements IContentOutlinePage
{
   private Canvas overview;
   private ScalableFreeformRootEditPart rootEditPart;

   private Thumbnail thumbnail;

   public OverviewOutlinePage(ScalableFreeformRootEditPart rootEditPart)
   {
      super();
      this.rootEditPart = rootEditPart;
   }

   public void addSelectionChangedListener(ISelectionChangedListener listener)
   {}

   public void createControl(Composite parent)
   {
      //创建与LightweightSystem交互的Canvas
      overview = new Canvas(parent, SWT.NONE);
      LightweightSystem lws = new LightweightSystem(overview);

      //创建编辑器的缩略图
      thumbnail =
         new ScrollableThumbnail((Viewport) rootEditPart.getFigure());
      thumbnail.setBorder(new MarginBorder(3));
      thumbnail.setSource(
         rootEditPart.getLayer(LayerConstants.PRINTABLE_LAYERS));
      //在LightweightSystem中添加缩略图
      lws.setContents(thumbnail);
   }

   public void dispose()
   {
      if (null != thumbnail)
         thumbnail.deactivate();

      super.dispose();
   }

   public Control getControl()
   {
      return overview;
   }
```

```
        public ISelection getSelection()
        {
            return StructuredSelection.EMPTY;
        }

        public void removeSelectionChangedListener(ISelectionChangedListener listener)
        {}

        public void setFocus()
        {
            if (getControl() != null)
                getControl().setFocus();
        }

        public void setSelection(ISelection selection)
        {}
    }
```

如上例所示，大纲主要通过 createControl 方法创建了显示的 SWT 组件，以及通过 LightweightSystem 把 SWT 组件和 Draw2d 的 Thumbnail（缩略图）关联起来。

21.10 小　　结

本章介绍了一个表模型编辑器的实现，程序实现了 GEF 编辑器所具备的基本功能，另外在源程序中还实现了编辑模型的校验、鼠标右键菜单的实现和模型重新布局(单击工具栏按钮)等功能，读者在看完本章后仔细推敲源代码，必定会有很大的收获。

第 22 章　EMF 的使用

EMF（Eclipse Modelling Framework）是 Eclipse 中的建模框架，EMF 自发布以来一直受到 Eclipse 社区的热情拥护，目前很多 Eclipse 的子项目都是基于它开发的。本章将介绍 EMF 及通过 EMF 来生成模型和编辑器。

本章的内容包括如下几个方面。

- EMF 的简介
- 建立一个 EMF 应用
- 使用 EMF 生成 GEF 的模型

22.1　EMF 的简介

EMF 是用来定义和实现结构化模型的框架，这些数据模型是一系列在程序中处理数据的相关的类。EMF 是 Eclipse 中的建模框架，要完全掌握 EMF 应该对模型驱动开发（MDA）有一定的了解，而 EMF 可以看做是 Eclipse 上 MDA 的一个实现（代码生成是 MDA 的重要组成部分之一），它能够生成在 Eclipse 上执行的代码。

MDA 讲究的是把模型和应用系统实现分开，模型是最重要的部分，可以说有了清楚的模型，就完成了一半的工作。在 EMF 中，模型是由元模型（Meta Model）定义的，例如 UML 里“类”和“属性”这些概念是在 UML 的元模型里定义的，而元模型又由“元元模型”来定义，后者多是自描述的，也就是能够自己定义自己，所以很少见到“元元元模型”的概念。EMF 定义了一套 Ecore 元模型，这是一个自描述的模型。用 Ecore 元模型可以定义 ecore 模型，也就是.ecore 文件。

在 MDA 中，UML 图可以帮助我们直观地了解应用系统里各对象的关系，但在开发过程中，UML 图里的定义很可能被修改，如果这一修改没有及时反映到 UML 图，UML 图就会逐渐变得不准确而失去作用。但是保持代码和 UML 图的一致是一件很烦琐的工作，有时由于项目管理的需要，UML 图又必须保证能够反映系统的真实结构。比较好的解决方法是让代码由 UML 图直接生成，模型需要修改时也在 UML 图上做改动，并且重新生成代码，这正是 EMF 的专长。

在 Eclipse 中使用 EMF 有如下几方面的好处。

（1）代码生成：所有基于模型的代码都可以自动生成，EMF 生成的代码都是从模板文件生成，且可读性比较好的代码。另外，用户生成的代码是可定制的，且用户在生成代码的基础上所做的修改不会因为重新生成而丢失。

（2）输入：用户能通过模型定制文件作为代码生成的输入文件，包括“Rational Rose”的模型文件、Java 的接口文件和 XML Schema 等。

（3）默认的持久化机制：EMF 能够保存和加载 XML 的模型文件，这些模型文件是 XML 格式的，如果用户想把模型持久化成其他的格式，在 EMF 也是允许的。

（4）模型编辑器：EMF 能够为用户生成模型编辑器，通过编辑器用户能够查看和编辑模型文件，另外，用户还能定义模型编辑器。

22.2 使用 EMF 生成模型

EMF 能够从“Rational Rose”文件、Java 的接口文件和 XML Schema 等生成相关的代码，在生成代码之前，要把这些文件转换成 EMF 认识的文件格式（genmodel）。本节将通过定制一个简单的 UML 图，在此基础上生成模型代码。

22.2.1 定制类关系图

通过建模工具，用户能够直观地编辑模型及模型之间的关系。建模工具有“Rational Rose”、“Together”等，本节通过 Omondo 公司的 EclipseUML 插件来画类关系图。“EclipseUML Free Edition”版是免费的，且功能基本上能满足用户通常建模的需要。

要通过 EclipseUML 进行建模，先要到 Omondo 网站“http://www.omondo.com”下载 EclipseUML 插件，并安装。下载下来的 EclipseUML 是一个可直接运行的 jar 文件，双击 jar 文件可直接运行，运行后指定 Eclipse 的安装目录，按向导执行就可以安装好 EclipseUML 了，如图 22-1 所示。

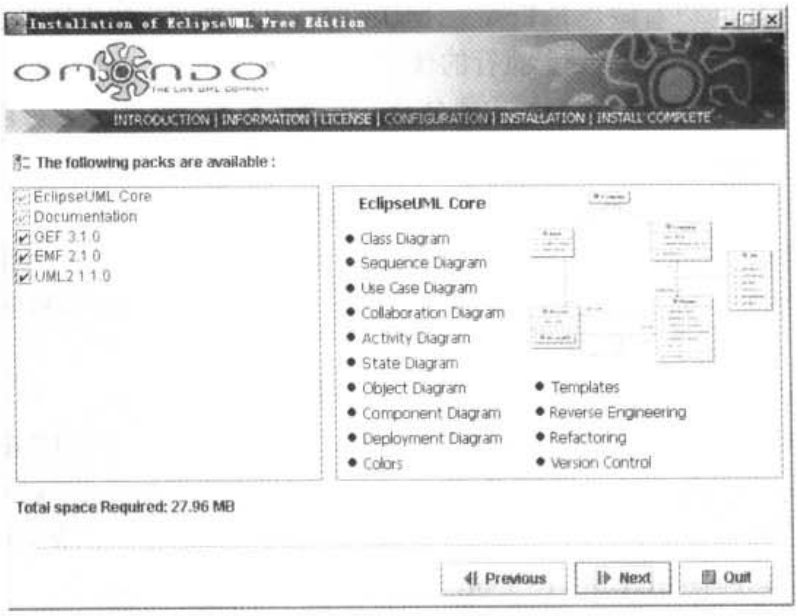


图 22-1 安装 EclipseUML

安装 EclipseUML 后，启动安装 EclipseUML 的 Eclipse 就可以通过 EclipseUML 定制所需的类关系图了，步骤如下。

（1）通过向导创建空的 EMF 项目（Empty EMF project）。

（2）通过新建向导创建一个 EclipseUML 的类图（EMF Class Diagram）。

（3）在“Class Diagram”上添加类、属性和关系等，如图 22-2 所示。

（4）新建 EMF 模型向导，生成 genmodel。首先通过新建向导选择 EMF 模型（EMF Model

项），其次选择要通过 Ecore 模型生成（Ecore model 项），最后在向导中选择上一步类图中生成的 ecore 文件，如图 22-3 所示。

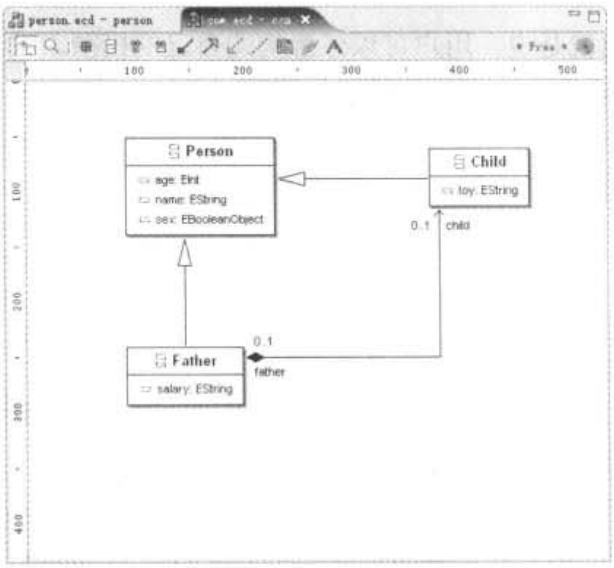


图 22-2　类关系图

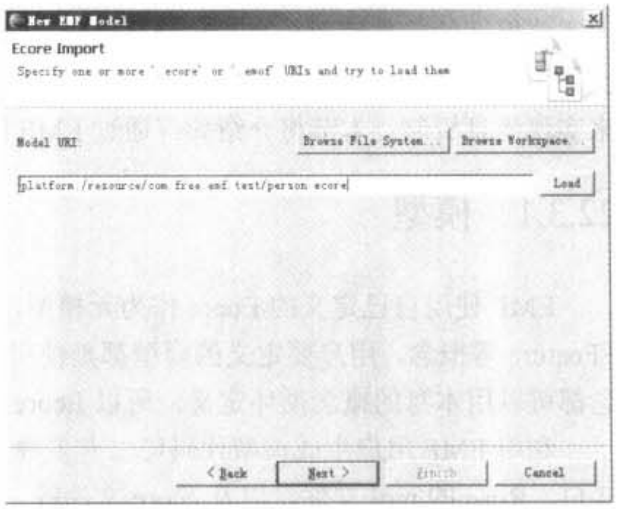


图 22-3　选择 ecore 文件

上面的步骤通过 ecore 文件生成了一个 genmodel 文件，在 genmodel 模型的基础上还可以做一些定制工作，例如每个属性的描述信息等。

22.2.2　生成模型代码

有了 genmodel 文件，生成代码是非常容易的事。打开 genmodel 文件，在打开编辑器的根节点下单击鼠标右键，选择“Generate Model Code”（生成模型代码），如图 22-4 所示。



图 22-4　生成代码

本节只利用类图生成 EMF 对应的模型，所以选择“Generate Model Code”（生成模型代码）项即可。另外，读者可以选择其他的生成项，EMF 也可生成编辑器和测试的相关代码。选择生成模型代码后，在当前项目的源代码中会添加相应的模型代码。

22.3 在 GEF 框架中使用 EMF 模型

GEF 的设计没有对模型部分做任何限制，也就是说，用户可以任意构造自己的模型，唯一需要保证的就是模型具有某种消息机制，以便在发生变化时能够通知视图（通过 EditPart）。在前面几章，作者都是利用 java.beans 包中的 PropertyChangeSupport 和 PropertyChangeListener 来实现消息机制。本节将介绍如何通过 EMF 构造 GEF 模型，以及实现消息机制。

22.3.1 模型

EMF 使用自己定义的 Ecore 作为元模型，在这个元模型里定义了 EPackage、EClassifier、EFeature 等概念，用户要定义的模型都是使用这些概念来定义的。同时因为 Ecore 中的所有概念都可以用本身的概念循环定义，所以 Ecore 又是自己的元模型，也就是元元模型。

利用 EMF 用户生成模型代码可以有多种方式，例如通过 XML Schema、带有注释的 Java 接口、Rose 的 mdl 文件，以及.ecore 文件等。EMF 的代码生成器需要一个扩展名为.genmodel 的文件提供信息，这个文件可以通过上面说的几种方式生成。本节的模型类图如图 22-5 所示。

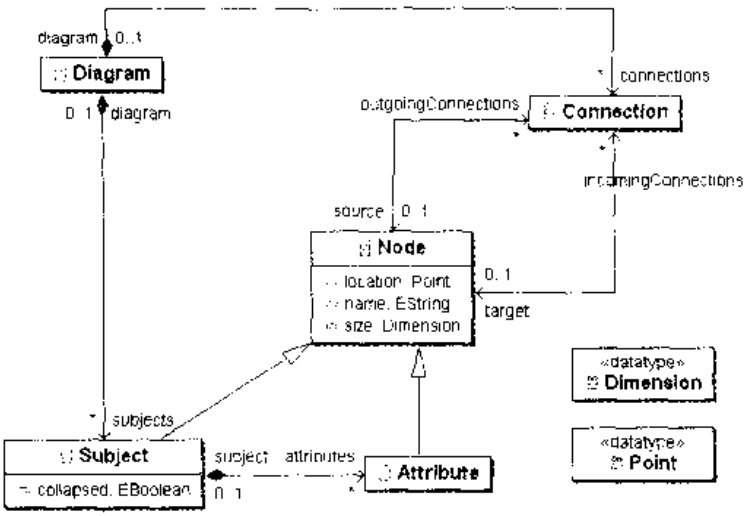


图 22-5 模型类图

为了让模型与编辑器更好地分离，在此让 EMF 模型单独位于一个 Plugin 中（名为 SubjectModel），而让编辑器 Plugin（SubjectEditor）依赖于它。这样做的另一个好处是，当修改模型后，如果用户愿意，可以很容易地删除以前生成的代码，然后全部重新生成。

生成的模型都是 Notifier 对象，它维护了一个 Adapter 列表，可以把 Adapter 作为监听器加入到模型的这个列表中。Notifier 对象相当于 PropertyChangeSupport，监听器可以把自己注册到模型中。

22.3.2 EditPart 的修改

在前面几章介绍 GEF 时知道，EditPart 是实现 java.beans.PropertyChangeListener 接口的，当模型改用 EMF 实现后，EditPart 应改为实现 org.eclipse.emf.common.notify.Adapter 接口，因

为 EMF 的每个模型对象都是 Notifier，它维护了一个 Adapter 列表，可以把 Adapter 作为监听器加入到模型的这个列表中。

实现 Adapter 接口时需要实现 getTarget()和 setTarget()方法，target 代表发出消息的那个模型对象。此实现方式是在 EditPart 里维护一个 Notifier 类型的 target 变量，这两个方法分别返回和设置该变量即可。

另外，还要实现 isAdapterForType()方法，该方法返回一个布尔值，表示这个 Adapter 是否应响应指定类型的消息，在此实现一律为“return type.equals(getModel().getClass());”。

另外，propertyChanged()方法的名称应改为 notifyChanged，其实现的功能和以前是一样的，但代码有所不同。下面代码是在 NodePart 中的实现。

```
public void notifyChanged(Notification notification) {
    int featureId = notification.getFeatureID(ModelPackage.class);
    switch (featureId) {
    case ModelPackage.NODE__LOCATION:
    case ModelPackage.NODE__SIZE:
        refreshVisuals();
        break;
    case ModelPackage.NODE__INCOMING_CONNECTIONS:
        refreshTargetConnections();
        break;
    case ModelPackage.NODE__OUTGOING_CONNECTIONS:
        refreshSourceConnections();
        break;
    }
}
```

事件通知的方式和前面介绍的 propertyChanged 方法类似，但通知的标识是 int 型的变量，而用 PropertyChangeSupport 时是采用一个字符串的 ID 值。

另外，在 EditPart 还有 active()/deactive()方法中的内容需要修改，作用还是把 EditPart 自己作为 Adapter（不是 PropertyChangeListener 了）加入模型的监听器列表。下面是 SubjectPart 的实现，其中 eAdapters()得到监听器列表。

```
public void activate() {
   super.activate();
   getSubject().eAdapters().add(this);
}

public void deactivate() {
   super.deactivate();
   getSubject().eAdapters().remove(this);
}
```

从上面的代码可以看出，改成 EMF 模型后，EditPart 所做的修改实际是在两种消息机制之间的转换，如果对以前的那套机制很熟悉的话，这里理解起来不应该有任何困难。

22.3.3 ElementFactory 的修改

这个类的作用是根据 template 创建新的模型对象实例，以前的实现都是“new XXX()”这

样的，用了 EMF 以后应改为“ModelFactory.eINSTANCE.createXXX()”。EMF 里的每个模型对象实例都应该是使用工厂创建的，代码如例程 22-1 所示。

例程 22-1 ElementFactory.java

```
package com.test.subject.model;

import org.eclipse.emf.ecore.EPackage;
import org.eclipse.gef.requests.CreationFactory;

public class ElementFactory implements CreationFactory {

   private Object template;

   public ElementFactory(Object template) {
      this.template = template;
   }

   public Object getNewObject() {
      if (template.equals(Diagram.class))
         return ModelFactory.eINSTANCE.createDiagram();
      else if (template.equals(Subject.class))
         return ModelFactory.eINSTANCE.createSubject();
      else if (template.equals(Attribute.class))
         return ModelFactory.eINSTANCE.createAttribute();
      else if (template.equals(Connection.class))
         return ModelFactory.eINSTANCE.createConnection();
      return null;
   }

   public Object getObjectType() {
      return template;
   }
}
```

在原先的创建选项板中定义 CreationEntry 时指定 SimpleFactory 作为工厂，这个类是使用 Class.newInstance()创建新的对象实例。而用 EMF 作为模型后，创建实例的工作应该交给 ModelFactory 来完成，所以必须定义自己的 CreationFactory。ElementFactory 就实现了如此的功能。

22.3.4 处理自定义数据类型

在 Node 类里有两个非标准数据类型：Point 和 Dimension。要让 EMF 能够正确地将它们保存，必须提供序列化和反序列化它们的方法。在 EMF 生成的代码里，找到 ModelFactoryImpl 类，这里有形如 convertXXXToString()和 createXXXFromString()的几个方法，分别用来序列化和反序列化这种外部数据类型。用户要把它的默认实现改为自己的方式。下面是 Point 的实现方式。

```
public String convertPointToString(EDataType eDataType, Object instanceValue) {
   Point p = (Point) instanceValue;
```

```
        return p.x + "," + p.y;
    }

    public Point createPointFromString(EDataType eDataType, String initialValue) {
        Point p = new Point();
        String[] values = initialValue.split(",");
        p.x = Integer.parseInt(values[0]);
        p.y = Integer.parseInt(values[1]);
        return p;

    }
```



修改后要将方法前面的@generated注释删除，这样在重新生成代码时才不会被覆盖掉。要设置使用这些类型的变量的默认值会有点问题（例如设置Node类的location属性的默认值），在EMF自带的Sample Ecore Model Editor里设置它的defaultValueLiteral为“100，100”（这是我们通过convertPointToString()方法定义的序列化形式）会报一个错，但不管它就可以了，在生成的代码里会得到这个默认值。

22.3.5 保存和载入模型

EMF通过Resource管理模型数据，几个Resource放在一起称为ResourceSet。前面说过，要想正常保存模型，必须保证每个模型对象都被包含在Resource里，当然间接包含也是可以的。例如例子中的这个模型，Diagram是被包含在Resource里的（创建新Diagram时即被加入），而Diagram包含Subject，Subject包含Attribute，所以它们都在Resource里。在类图中可以看到，Diagram和Connection之间存在一对多的包含关系，这个关系的主要作用就是确保在保存模型时不会出现DanglingHREFException，因为如果没有这个包含关系，则Connection对象不会被包含在任何Resource里。

在删除一个对象的时候，一定要保证它不再包含在Resource里，否则保存后的文件中会出现很多空元素。比较容易犯错的地方是对Connection的处理，在删除连接的时候，只是从源节点和目标节点里删除对这个连接的引用是不够的，因为这样只是在界面上消除了两个节点间的连接线，而这个连接对象还是包含在Diagram里的，所以还要从Diagram对象里删除它才对。DeleteConnectionCommand中执行的代码如下。

```
    public void execute() {
        source.getOutgoingConnections().remove(connection);
        target.getIncomingConnections().remove(connection);
        connection.getDiagram().getConnections().remove(connection);
    }
```

当然，新建连接时也不要忘记将连接添加在Diagram对象里（代码见CreateConnectionCommand）。保存和载入模型的代码请看SubjectEditor的init()方法和doSave()方法，都是很标准的EMF访问资源的方法。以下是载入的代码（如果是新创建的文件，则在Resource中新建Diagram对象）。

```
public void init(IEditorSite site, IEditorInput input) throws PartInitException {
    super.init(site, input);
    IFile file = ((FileEditorInput) getEditorInput()).getFile();
    URI fileURI = URI.createPlatformResourceURI(file.getFullPath().toString());
    resource = new XMLResourceImpl(fileURI);
    try {
        resource.load(null);
        diagram = (Diagram) resource.getContents().get(0);
    } catch (IOException e) {
        diagram = ModelFactory.eINSTANCE.createDiagram();
        resource.getContents().add(diagram);
    }
}
```

程序运行效果如图 22-6 所示。

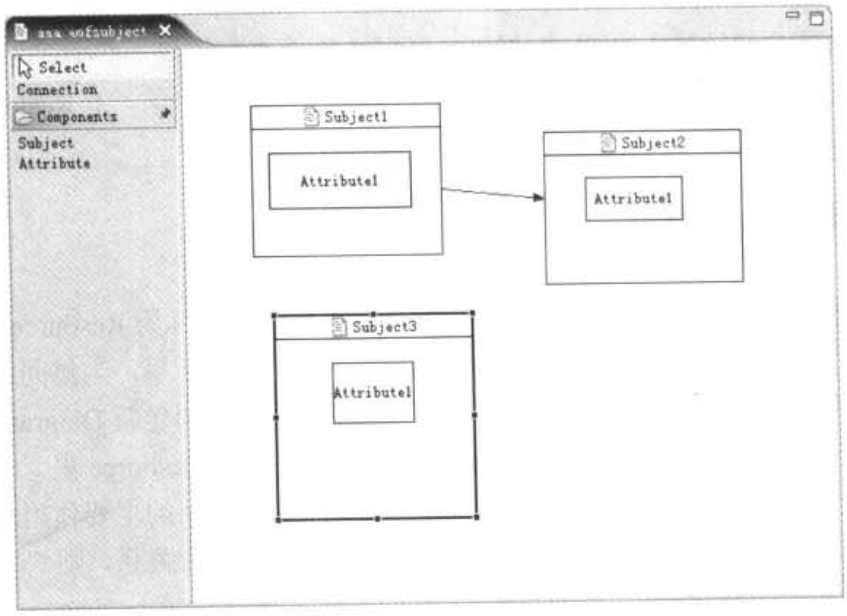


图 22-6　使用 EMF 模型创建 GEF 编辑器

通过 EMF 能生成模型，在 GEF 框架中也可以利用 EMF 模型，但 EMF 和 GEF 之间不存在依赖关系。另外，EMF 也能生成相应的模型编辑器，但不是 GEF 编辑器。



GMF 框架结合了 Eclipse 中的 GEF 和 EMF 框架，可以通过 GMF 快速生成 GEF 编辑器，另外还可以通过开源插件 Merlin 在 EMF 的基础上直接生成 GEF 编辑器。

22.4　小　　结

本节简要介绍了如何生成 EMF 模型，以及如何在 GEF 框架中使用 EMF 模型。但 EMF 是一个比较成熟的框架，它的功能非常强大，不是一两章就能介绍清楚的。本章旨在作为一个 EMF 入门，让读者能对 EMF 稍有了解。

第 23 章　RCP 技术

Java 在 B/S 模式的开发中已经非常成熟，但在 C/S 模式的开发中，还是远落后于 VC、Delphi 等开发语言。RCP（Rich Client Platform）是 Eclipse 中为创建富客户程序而贡献的一种技术，通过 RCP 技术开发出来的程序可以与任何其他语言开发的本地程序相媲美。

本章将介绍 Eclipse 中 RCP 应用程序的实现，以及如何定制用户的 RCP 应用程序，包括如下几方面的内容。

- RCP 的简介
- 建立一个 RCP 的应用
- RCP 的组成部分

23.1　RCP 的简介

RCP 应用程序是基于 Eclipse 插件架构的，把 Eclipse 作为 Java 程序的宿主平台，在 Eclipse 的基础之上建立跨平台的 Java 应用，通过 RCP 技术开发出来的应用程序，可以与任何其他语言开发的本地程序相媲美。

在 Eclipse 的早期版本中，IDE 的菜单、标题和状态栏等都是“硬”编码到 Eclipse 中的。这种方式对 IDE 来说没太大影响，但如果要把 Eclipse 作为一个非 IDE 的程序，就会带来很多的问题。在 Eclipse 3.0 以后，Eclipse 已经被重构，所有“硬”编码的部分被分离出来，使得在 Eclipse 中 RCP 的开发成为可能。

在开发 RCP 应用之前，首先要了解几个概念。

- Applications（应用）：Applications 是 RCP 程序的入口，它相当于程序的控制器。Applications 的结构比较类似，用户要通过“org.eclipse.core.runtime.applications”扩展点实现 RCP 应用。
- Workbench（工作台）和 Workbench Windows（工作台窗口）：Workbench 的功能是维护 RCP 框架，在一个应用中只有一个工作台，但可以有多个工作台窗口。

23.2　建立一个 RCP 的应用

PDE 提供了创建 RCP 的快捷方式，用户可以通过向导，利用 PDE 提供的模板快速地创建一个 RCP 应用。本节将通过向导创建 RCP 程序，运行 RCP 程序，最后把 RCP 程序导出成可单独运行的应用。通过这个过程，读者可以了解 RCP 程序的强大之处。

23.2.1 创建 RCP 应用

在 Eclipse 中，用户可以通过 PDE 快速地创建一个 RCP 应用。创建 RCP 应用和创建插件类似（RCP 也是基于插件架构的应用）。通过模板创建 RCP 应用的步骤如下。

（1）选择新建插件项目，并在新建插件向导中输入插件项目的名称。

（2）在设置插件属性的对话框中选择创建 RCP 应用，如图 23-1 所示。

（3）单击“Next”按钮，选择 RCP 应用的模板，如图 23-2 所示。



图 23-1　设置插件的属性



图 23-2　选择 RCP 应用

（4）单击“Finish”按钮，通过模板创建 RCP 应用。

23.2.2 运行 RCP 应用

按照上面的步骤，可以通过模板非常容易地创建可独立运行的 RCP 应用，在此应用中实现了“org.eclipse.core.runtime.applications”和“org.eclipse.core.runtime.products”扩展点，用户可以在插件的运行参数设置对话框中设置以 application 或 product 方式运行。

在 Eclipse 中，运行 RCP 的步骤如下。

（1）执行菜单“Run”→“Debug”，在打开的对话框中设置运行参数，其中“Run a product”参数要指定 RCP 项目中的产品 ID，如图 23-3 所示。

（2）单击“Debug”按钮，运行 product 指定的插件项目，运行效果如图 23-4 所示。

从运行效果看，是不是感觉到所运行的 RCP 插件没有任何 Eclipse 的身影？在运行的 RCP 产品中只包含 Eclipse 的核心包和 SWT 等构建 GUI 所依赖的包，可以说 RCP 只是包括 Eclipse 内核的 Java 应用，这也是 RCP 程序开发的目的所在。

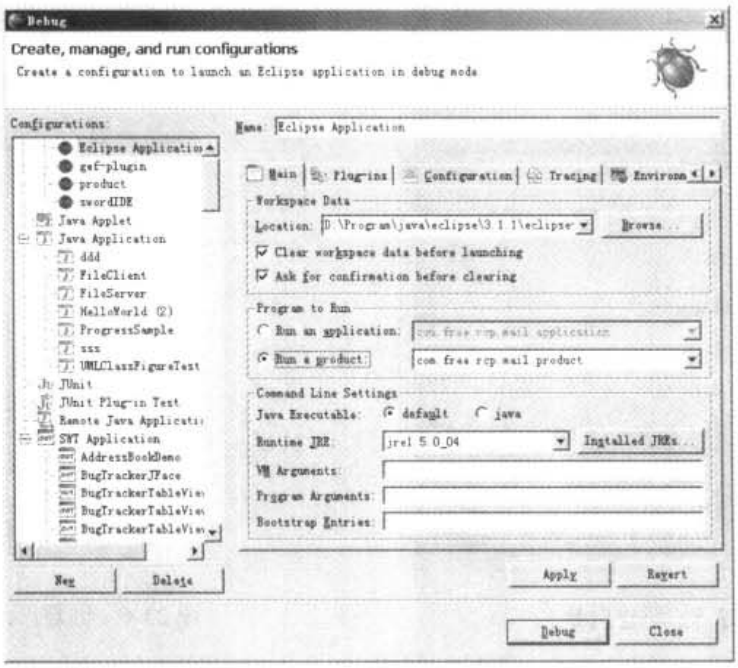


图 23-3 设置运行参数



图 23-4 RCP 范例运行效果

23.2.3 导出 RCP 应用

通过模板可以快速地创建一个 RCP 应用，用户可以在此基础上扩展，建立自己的可跨平台的 Java 应用。当创建了自己的应用以后，用户会考虑把创建的应用作为一个独立的产品进行发布，让其他人可以直接运行程序（就像用 VC 等开发出来的程序一样）。

Eclipse 的 PDE 提供了导出产品的功能，用户可通过 PDE 的向导，把自己开发的应用导出成为一个含 Eclipse 内核的、可执行的应用，具体步骤如下。

（1）通过菜单“File”→“New”→“Other”，打开新建向导，并选择“Product Configuration”项，创建产品的配置文件（通过产品配置文件，Eclipse 才能知道要打包产品包含插件的内容），如图 23-5 所示。

（2）输入产品的配置文件名称，并选择存在的产品，如图 23-6 所示。

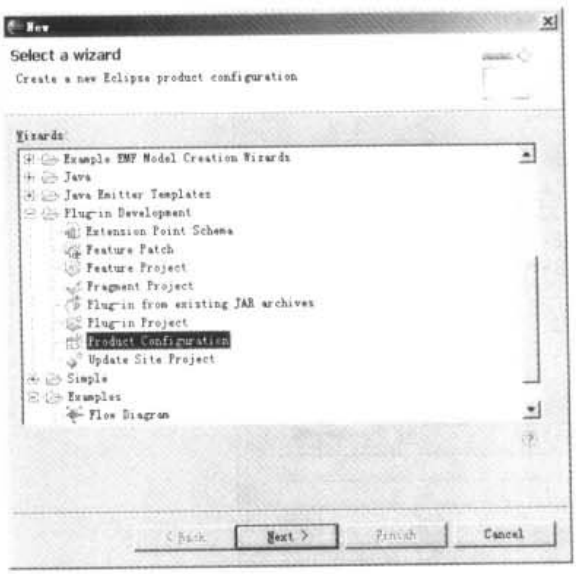


图 23-5　新建向导



图 23-6　创建产品配置文件

（3）单击“Finish”按钮，在当前的项目中创建了 mail.product 产品配置文件，用户可以通过产品配置文件的编辑器修改要打包发布产品的一些配置信息，例如“可执行程序名字”、“图标”和“欢迎界面”等。

（4）用鼠标右键单击要导出的项目，选择“Export”菜单，打开 Export 对话框，并选择“Eclipse Product”选项，如图 23-7 所示。

（5）单击“Next”按钮，并在导出 RCP 项目的对话框中设置相关的属性，通过“Browse”按钮选择产品配置文件，以及指定导出的目录等，如图 23-8 所示。



图 23-7　Export 对话框



图 23-8　导出产品对话框

通过上面的步骤就可以把创建的 RCP 应用导出到“D:\temp\rcp”目录下面，产品根目录为 mail。

导出的 RCP 应用是一个包含 Eclipse 内核的应用，在此应用中只包含用户开发的插件所依赖的插件和内核，所以导出项目总的磁盘空间是很小的，不会导出 Java 的 IDE 和其他非依赖的插件。RCP 项目中的插件列表如图 23-9 所示。

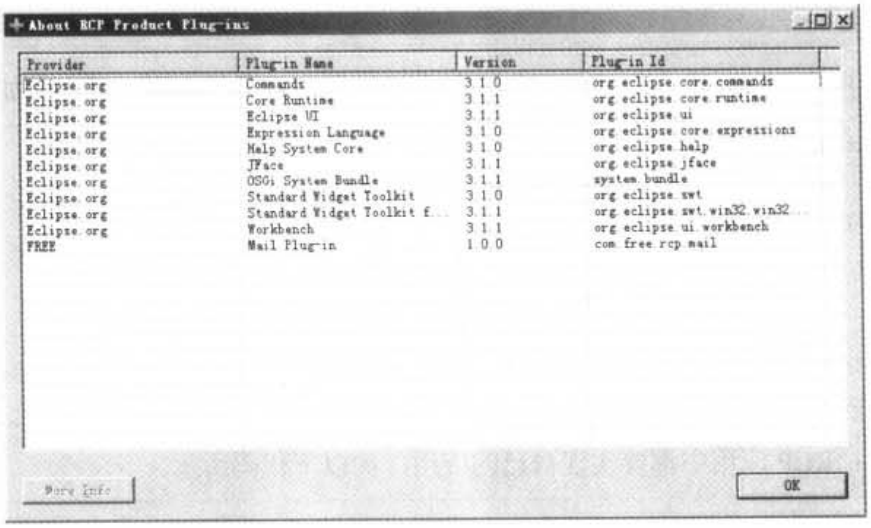


图 23-9 RCP 项目插件列表

23.3 RCP 的组成部分

通过 PDE 的向导，用户能很容易地创建 RCP 的应用，但如果要在此基础上继续扩展，则要了解 RCP 的组成及各部分的功能。本节将通过上节的 mail 实例介绍 RCP 应用的各个部分的功能。

23.3.1 Plugin.xml

RCP 程序是基于插件架构的，所以开发 RCP 程序首先是开发 Eclipse 插件，把用户开发的插件集成到 Eclipse 的内核中，共同组成 RCP 应用，RCP 程序也是通过 Plugin.xml 文件描述插件的。在上一节的 mail 示例中，Plugin.xml 文件描述了实现扩展点的几个部分，介绍如下。

（1）视图：mail 应用定义了两个视图，一个为 Message 视图，另一个为 Mailboxes 视图，如以下代码所示。

```
<extension
      point="org.eclipse.ui.views">
   <view
        name="Message"
        allowMultiple="true"
        icon="icons/sample2.gif"
        class="com.free.rcp.mail.View"
        id="com.free.rcp.mail.view">
   </view>
   <view
        name="Mailboxes"
        allowMultiple="true"
        icon="icons/sample3.gif"
        class="com.free.rcp.mail.NavigationView"
        id="com.free.rcp.mail.navigationView">
   </view>
```

```
</extension>
```

（2）默认视角：在 mail 应用中，视角只包含了应用中定义的视图，其定义如以下代码所示。

```
<extension
      point="org.eclipse.ui.perspectives">
   <perspective
       name="Mail Perspective"
       class="com.free.rcp.mail.Perspective"
       id="com.free.rcp.mail.perspective">
   </perspective>
</extension>
```

（3）应用：RCP 应用中都要实现自己的应用，如以下代码所示。

```
<extension
     id="application"
     point="org.eclipse.core.runtime.applications">
   <application>
     <run
         class="com.free.rcp.mail.Application">
     </run>
   </application>
</extension>
```

（4）产品：可以通过产品来发布用户定义的应用，如以下代码所示。

```
<extension
      id="product"
      point="org.eclipse.core.runtime.products">
   <product
       application="com.free.rcp.mail.application"
       name="RCP Product">
     <property
         name="aboutText"
         value="%aboutText">
     </property>
     <property
         name="windowImages"
         value="icons/sample2.gif">
     </property>
     <property
         name="aboutImage"
         value="product_lg.gif">
     </property>
   </product>
</extension>
```

mail 示例是一个比较完整的 RCP 应用，用户可以看到，mail 实现了自己的视图、视角和产品的扩展点，组织应用的布局和相关的实现。下面将介绍 RCP 应用中所实现的功能。

23.3.2 应用的入口 Application

Eclipse 的运行时组件通过“org.eclipse.core.runtime.applications”扩展点找到 RCP 应用的入口，执行应用。应用扩展点要实现 IPlatformRunnable 接口，Eclipse 会执行应用实现类的 run 方法，代码如例程 23-1 所示。

例程 23-1 Application.java

```
package com.free.rcp.mail;

import org.eclipse.core.runtime.IPlatformRunnable;
import org.eclipse.swt.widgets.Display;
import org.eclipse.ui.PlatformUI;

public class Application implements IPlatformRunnable {

    public Object run(Object args) throws Exception {
        //得到display
        Display display = PlatformUI.createDisplay();
        try {
            //创建工作台
            int returnCode = PlatformUI.createAndRunWorkbench(display,
                new ApplicationWorkbenchAdvisor());
            if (returnCode == PlatformUI.RETURN_RESTART) {
                return IPlatformRunnable.EXIT_RESTART;
            }
            return IPlatformRunnable.EXIT_OK;
        } finally {
            display.dispose();
        }
    }
}
```

Application 类实现了 IPlatformRunnable 接口，Eclipse 内核会执行 run 方法运行应用。在 run 方法中，通过 PlatformUI 类的 createAndRunWorkbench 方法创建了应用的工作台。

23.3.3 工作台 ApplicationWorkbenchAdvisor

Application 通过 PlatformUI 的 createAndRunWorkbench 创建应用的工作台，其中第二个参数为 WorkbenchAdvisor，表示工作台的配置等。例如可以获得应用的视角，代码如例程 23-2 所示。

例程 23-2 ApplicationWorkbenchAdvisor.java

```
package com.free.rcp.mail;

import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchAdvisor;
import org.eclipse.ui.application.WorkbenchWindowAdvisor;
```

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

    //视角 ID
    private static final String PERSPECTIVE_ID = "com.free.rcp.mail.perspective";

    //创建工作台窗口
    public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor(
            IWorkbenchWindowConfigurer configurer) {
        return new ApplicationWorkbenchWindowAdvisor(configurer);
    }

    //获得当前工作台的视角
    public String getInitialWindowPerspectiveId() {
        return PERSPECTIVE_ID;
    }
}
```

如以上代码所示，通过 ApplicationWorkbenchAdvisor 类的 getInitialWindowPerspectiveId 可以获得当前工作台的视角，通过 createWorkbenchWindowAdvisor 方法能创建当前工作台的窗口。

另外，用户可以通过实现 WorkbenchAdvisor 类，以及重写 WorkbenchAdvisor 的方法，控制工作台的生命周期中每个切入点的执行方法，主要有如下的方法。

（1）工作台生命周期相关方法

WorkbenchAdvisor 类有几个方法和工作台的生命周期有关，用户能通过重写这几个方法，在工作台生命周期的某个特定点执行自己的代码，如表 23-1 所示。

表 23-1　工作台生命周期相关方法

方　法	描　述	参　数
initialize	工作台初始化	IWorkbenchConfigurer
preStartup	在 initialize 之后，工作台窗口被打开之前执行	无
postStartup	在所有的工作台窗口被打开之后执行	无
preShutdown	工作台被停止之前（窗口被关闭之前）执行	无
postShutdown	工作台被停止之后（窗口被关闭之后）执行	无

（2）消息循环相关方法

消息循环负责处理用户的输入，然后把输入分配到相关的监听器进行处理，WorkbenchAdvisor 中提供了一些钩子方法，处理消息循环的某些问题，如表 23-2 所示。

表 23-2　消息循环相关方法

方　法	描　述	参　数
eventLoopException	当消息循环中有未处理的异常时执行	Throwable
eventLoopIdle	当消息队列中无处理的消息时执行	Display

（3）配置相关方法

WorkbenchAdvisor 中能通过重写一些方法，实现工作台的配置，其中最常用的是 getInitialWindowPerspectiveId 方法，返回视角的 ID。

23.3.4 工作台窗口 ApplicationWorkbenchWindowAdvisor

ApplicationWorkbenchAdvisor 类通过 createWorkbenchWindowAdvisor 方法能创建工作台窗口。通过实现 WorkbenchWindowAdvisor 类，用户能定制工作台窗口的大小、状态栏、工具栏等信息，代码如例程 23-3 所示。

例程 23-3 ApplicationWorkbenchWindowAdvisor.java

```
package com.free.rcp.mail;

import org.eclipse.swt.graphics.Point;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchWindowAdvisor;

public class ApplicationWorkbenchWindowAdvisor extends WorkbenchWindowAdvisor {

    public ApplicationWorkbenchWindowAdvisor(IWorkbenchWindowConfigurer configurer) {
        super(configurer);
    }

    //创建Action Bar
    public ActionBarAdvisor createActionBarAdvisor(IActionBarConfigurer configurer) {
        return new ApplicationActionBarAdvisor(configurer);
    }

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
        //设置工作台窗口的大小
        configurer.setInitialSize(new Point(600, 400));
        //是否显示CoolBar
        configurer.setShowCoolBar(true);
        //是否显示状态栏
        configurer.setShowStatusLine(false);
    }
}
```

用户也可以通过重写 WorkbenchWindowAdvisor 中的方法，控制工作台窗口生命周期的执行过程。例如，上例的 preWindowOpen 方法，在窗口打开直接设置窗口的大小、是否显示状态栏等属性，这些方法大多数以“pre”、“post”和“create”开始。

23.3.5 工作台“Action Bar” ApplicationActionBarAdvisor

在 Eclipse 中，“Action Bar”是菜单、工具栏和状态栏的统称，通过 ActionBarAdvisor 可以定制 RCP 应用的“Action Bar”，代码如例程 23-4 所示。

例程 23-4 ApplicationActionBarAdvisor.java

```
package com.free.rcp.mail;

import org.eclipse.jface.action.Action;
import org.eclipse.jface.action.GroupMarker;
import org.eclipse.jface.action.ICoolBarManager;
import org.eclipse.jface.action.IMenuManager;
import org.eclipse.jface.action.IToolBarManager;
import org.eclipse.jface.action.MenuManager;
import org.eclipse.jface.action.Separator;
import org.eclipse.jface.action.ToolBarContributionItem;
import org.eclipse.jface.action.ToolBarManager;
import org.eclipse.swt.SWT;
import org.eclipse.ui.IWorkbenchActionConstants;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.actions.ActionFactory;
import org.eclipse.ui.actions.ActionFactory.IWorkbenchAction;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;

public class ApplicationActionBarAdvisor extends ActionBarAdvisor {

    private IWorkbenchAction exitAction;
    private IWorkbenchAction aboutAction;
    private IWorkbenchAction newWindowAction;
    private OpenViewAction openViewAction;
    private Action messagePopupAction;


    public ApplicationActionBarAdvisor(IActionBarConfigurer configurer) {
        super(configurer);
    }

    protected void makeActions(final IWorkbenchWindow window) {

        //创建相应的Action
        exitAction = ActionFactory.QUIT.create(window);
        register(exitAction);

        aboutAction = ActionFactory.ABOUT.create(window);
        register(aboutAction);

        newWindowAction = ActionFactory.OPEN_NEW_WINDOW.create(window);
        register(newWindowAction);

        openViewAction = new OpenViewAction(window, "Open Another Message View", View.ID);
        register(openViewAction);

        messagePopupAction = new MessagePopupAction("Open Message", window);
        register(messagePopupAction);
```

```
    }

    protected void fillMenuBar(IMenuManager menuBar) {
     //添加文件和帮助菜单
       MenuManager fileMenu = new MenuManager("&File",
              IWorkbenchActionConstants.M_FILE);
       MenuManager helpMenu = new MenuManager("&Help",
              IWorkbenchActionConstants.M_HELP);

       menuBar.add(fileMenu);
       // Add a group marker indicating where action set menus will appear.
       menuBar.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
       menuBar.add(helpMenu);

       //添加文件菜单的Action
       fileMenu.add(newWindowAction);
       fileMenu.add(new Separator());
       fileMenu.add(messagePopupAction);
       fileMenu.add(openViewAction);
       fileMenu.add(new Separator());
       fileMenu.add(exitAction);

       //添加帮助菜单的Action
       helpMenu.add(aboutAction);
    }

    protected void fillCoolBar(ICoolBarManager coolBar) {
     //添加CoolBar
       IToolBarManager toolbar = new ToolBarManager(SWT.FLAT | SWT.RIGHT);
       //添加ToolBar到CoolBar中
       coolBar.add(new ToolBarContributionItem(toolbar, "main"));
       //添加Action到ToolBar中
       toolbar.add(openViewAction);
       toolbar.add(messagePopupAction);
    }
}
```

以上程序通过 makeActions 方法创建了所有的 Action，通过 fillMenuBar 方法添加了文件菜单和帮助菜单，通过 fillCoolBar 方法添加了 CoolBar 工具栏项。菜单和工具栏共用相关的 Action。

23.3.6 应用的视角（perspective）

WorkbenchAdvisor 中 getInitialWindowPerspectiveId 方法是一个抽象方法，也就是说要通过 WorkbenchAdvisor 创建工作台，用户必须实现自己的视角（也可以是 Eclipse 中现有的视角）。工作台会通过 getInitialWindowPerspectiveId 方法得到视角扩展点的 ID，并用此视角对视图布局。本例中视角的实现代码如例程 23-5 所示。

例程 23-5 Perspective.java

```
package com.free.rcp.mail;
```

```
import org.eclipse.ui.IFolderLayout;
import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;

public class Perspective implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout layout) {
        String editorArea = layout.getEditorArea();
        //设置编辑区域不可见
        layout.setEditorAreaVisible(false);

        //添加视图
        layout.addStandaloneView(NavigationView.ID,
                false, IPageLayout.LEFT, 0.25f, editorArea);
        //添加Folder，并在Folder中添加视图
        IFolderLayout folder = layout.createFolder("messages",
                IPageLayout.TOP, 0.5f, editorArea);
        folder.addPlaceholder(View.ID + ":*");
        folder.addView(View.ID);

        //设置NavigationView视图不可关闭
        layout.getViewLayout(NavigationView.ID).setCloseable(false);
    }
}
```

在 Perspective 视角中设置了编辑区域不可见，并添加了两个视图，其中 messages 视图放在一个 Folder 中，在此 Folder 中还能添加其他的视图，最后通过 setCloseable 方法设置 NavigationView 视图是不可关闭的。

在此，视图和其他扩展点的实现就不介绍了，读者可以参考本书源代码了解 mail 应用的实现。

23.4 小　　结

RCP 是 Eclipse 3.0 以后发展起来的，目的在于以 Eclipse 平台为框架，开发出基于 Java 的独立运行的应用。可以断言，随着 Eclipse 开发的不断深入和广为用户接受，RCP 程序将会在 Java 的桌面应用中占据一席之地。

本章介绍的内容不多，旨在让用户了解 RCP 应用实现流程。在本章，希望读者能够了解和 RCP 应用相关的几个类的实现，包括 IPlatformRunnable、WorkbenchAdvisor、WorkbenchWindowAdvisor 和 ActionBarAdvisor。

第 24 章　Eclipse 进阶

Eclipse 不仅仅是一个成功的 IDE，它还是一个可以尽情发挥的空间，用户可以不断地吸收它的养分，还可以尽情发挥自己的想象力。

本章将介绍如下几方面的内容。

- 选择与监听
- 命令堆栈
- 适配器
- UI 持久化
- 深入 Workbench

24.1　选择与监听

Eclipse 是一个功能强大的 UI 框架，它提供了很多可扩展的 UI 接口，用户可以通过这些接口扩展 UI 框架的功能，实现用户界面的可操作性。

回想一下 Eclipse 中的常用操作，当用户选择了某一个项目或文件时，Eclipse 的 Properties 视图将会显示和资源相关的属性，当选择了某个方法或属性时，大纲视图将会定位到相关的位置，如图 24-1 所示。



图 24-1　选择 Eclipse 项目节点

类似的功能是功能强大的 IDE 所应该具备的。然而 Eclipse 的视图和视图之间彼此独立，用户不太可能随时遍历每一个视图，判断某一个视图的状态，而且从效率上来说也是不可行的。Eclipse 通过选择服务（Selection Service）来解决这个问题，具体步骤如下。

（1）实现选择服务的提供者（ISelectionProvider）。

（2）获得 Eclipse 中的选择服务（ISelectionService）。

（3）实现选择服务的监听器（ISelectionListener）。

Eclipse 平台为用户的选择和监听提供了一种服务的机制，当选择的某个节点为可提供服务的对象时，工作台会把选择对象的动作作为事件通知监听器（可以是其他插件的其他视图），监听器监听到感兴趣的事件后，将会执行相关的操作（例如属性视图显示节点信息）。选择服务的通知机制如图 24-2 所示。

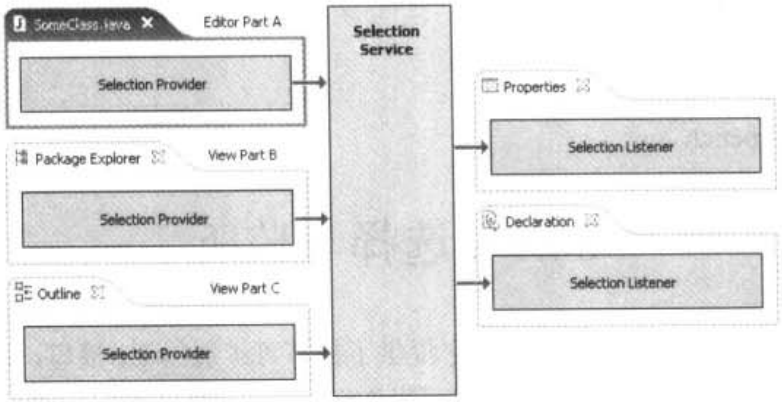


图 24-2　选择服务通知机制

24.1.1　选择事件提供者（Selection Provider）

在 Eclipse 中，被选择的对象都要实现 ISelectionProvider 接口，通过此接口提供选择其中子节点的能力，ISelectionProvider 接口定义如例程 24-1 所示。

例程 24-1　ISelectionProvider.java

```
package org.eclipse.jface.viewers;

public interface ISelectionProvider {
   public void addSelectionChangedListener(ISelectionChangedListener listener);

   public ISelection getSelection();

   public void removeSelectionChangedListener(
         ISelectionChangedListener listener);

   public void setSelection(ISelection selection);
}
```

ISelectionProvider 各方法解释如下。

- addSelectionChangedListener：添加选择节点的监听器。

- getSelection：获得当前的选择节点。
- removeSelectionChangedListener：删除选择节点的监听器。
- setSelection：设置选择节点。

JFace 的 Viewer 中都实现了 ISelectionProvider 接口，也就是说，用户可以监听到 Eclipse 中 JFace 的 Viewer 组件的选择事件。

实现 ISelectionProvider 接口的对象可以提供选择监听的能力，但是用户选择的并非实现 ISelectionProvider 接口的对象，而是其中的节点，例如 TreeViewer 中的节点项。能够被选择的对象是实现了 ISelection 接口的对象，所有可选择的对象都实现了 ISelection 接口。Eclipse 中一般认为有 3 种类型的选择对象，如图 24-3 所示。

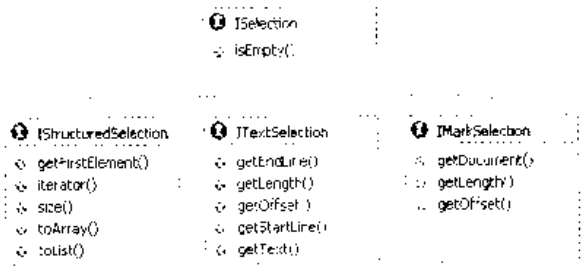


图 24-3 选择对象

ITextSelection、IStructuredSelection 和 IMarkSelection 分别表示文本类型的选择对象、结构化的选择对象和标记选择对象。用户通过监听器获得选择对象的内容。

24.1.2 选择服务（Selection Service）

工作台提供了追踪相关 Part 中节点被选择的服务，通过选择服务，用户可以注册选择的监听器。工作台提供选择服务由 ISelectionService 对象提供。ISelectionService 接口定义如例程 24-2 所示。

例程 24-2 ISelectionService.java

```
package org.eclipse.ui;

import org.eclipse.jface.viewers.ISelection;

public interface ISelectionService {
   public void addSelectionListener(ISelectionListener listener);

   public void addSelectionListener(String partId, ISelectionListener listener);

   public void addPostSelectionListener(ISelectionListener listener);

   public void addPostSelectionListener(String partId,
         ISelectionListener listener);

   public ISelection getSelection();

   public ISelection getSelection(String partId);
```

```
    public void removeSelectionListener(ISelectionListener listener);

    public void removeSelectionListener(String partId,
          ISelectionListener listener);

    public void removePostSelectionListener(ISelectionListener listener);

    public void removePostSelectionListener(String partId,
          ISelectionListener listener);
}
```

对用户来说，用户关心的只是什么节点被选择，以及如何处理节点被选择的事件，至于如何通知及什么时机通知比较适当，用户可以不用关心。Eclipse 已经封装了选择的服务层，用户可以通过工作台窗口获得 ISelectionService 对象，例如“getSite().getWorkbenchWindow().getSelectionService()”。ISelectionService 接口方法解释如下。

- addSelectionListener：添加选择的监听器，其中如果指定 partId，则表明监听指定的 Part。
- addPostSelectionListener：添加 Post 类型的选择监听器，其中如果指定 partId，则表明监听指定的 Part。
- getSelection：获得当前的选择节点。
- removeSelectionListener：删除选择监听器，其中如果指定 partId，则表明监听指定的 Part。
- removePostSelectionListener：删除 Post 类型的选择监听器，其中如果指定 partId，则表明监听指定的 Part。

Post 类型的选择监听器，在通过鼠标自动选择的场景是非常有用的，它会在一定的延时后才触发选择事件，而不会在选择某一项时触发选择事件，这样会大大提高效率。ISelectionService 对象能通过 addSelectionListener 和 addPostSelectionListener 方法添加监听器，例如“getSite().getWorkbenchWindow().getSelectionService().addSelectionListener(listener);”。

以上代码从 Site 中得到工作台窗口，通过工作台窗口得到 SelectionService，最后添加了选择监听器。

24.1.3 选择事件监听器（Selection Listener）

要实现选择事件的监听，就要实现选择监听器。选择监听器是一个 ISelectionListener 类型的对象。ISelectionListener 接口定义如例程 24-3 所示。

例程 24-3 ISelectionListener.java

```
package org.eclipse.ui;

import org.eclipse.jface.viewers.ISelection;

public interface ISelectionListener {
```

```
    public void selectionChanged(IWorkbenchPart part, ISelection selection);
}
```

如以上代码所示，ISelectionListener 只有一个方法，即 selectionChanged，当选择的对象改变后，将执行此过程。

选择监听器是用户比较关注的部分，通过 selectionChanged 方法，用户能知道哪一个 Part（IWorkbenchPart）中的哪一个对象（ISelection）被选用，从而进行相应的处理。如下代码片段为当选择的对象改变后，关联的 Action 也随之更新。

```
public void selectionChanged(IWorkbenchPart part, ISelection selection)
{
    updateActions(editPartActionIDs);
}
protected void updateActions(List actionIds)
{
    for (Iterator ids = actionIds.iterator(); ids.hasNext();)
    {
        IAction action = getActionRegistry().getAction(ids.next());
        if (null != action && action instanceof UpdateAction)
            ((UpdateAction) action).update();

    }
}
```

24.1.4 实例

Eclipse 中选择和监听是经常要用到的技术。下面将通过一个实例演示如何建立一个视图，然后通过视图监听 Eclipse 中的选择事件，以及把视图中的 Viewer 设置为选择提供者，代码如例程 24-4 所示。

例程 24-4 SelectionView.java

```
package com.mountainminds.eclipse.selectionsample;

import org.eclipse.jface.text.BadLocationException;
import org.eclipse.jface.text.Document;
import org.eclipse.jface.text.IMarkSelection;
import org.eclipse.jface.text.ITextSelection;
import org.eclipse.jface.text.TextViewer;
import org.eclipse.jface.viewers.ArrayContentProvider;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.model.WorkbenchLabelProvider;
import org.eclipse.ui.part.PageBook;
import org.eclipse.ui.part.ViewPart;
```

```
public class SelectionView extends ViewPart {

    //显示组件
    private PageBook pagebook;
    private TableViewer tableviewer;
    private TextViewer textviewer;

    //创建选择监听器
    private ISelectionListener listener = new ISelectionListener() {
        //响应选择改变事件
        public void selectionChanged(IWorkbenchPart sourcepart, ISelection selection) {
            //如果选择的 Part 不为当前视图，则在当前视图中显示选择的信息
            if (sourcepart != SelectionView.this) {
                showSelection(sourcepart, selection);
            }
        }
    };

    public void showSelection(IWorkbenchPart sourcepart, ISelection selection) {
        setContentDescription(sourcepart.getTitle() + " ("
          + selection.getClass().getName() + ")");
        //处理相应的选择类型
        if (selection instanceof IStructuredSelection) {
            IStructuredSelection ss = (IStructuredSelection) selection;
            //显示 IStructuredSelection 类型的选项
            showItems(ss.toArray());
        }
        if (selection instanceof ITextSelection) {
            ITextSelection ts  = (ITextSelection) selection;
            //显示 ITextSelection 类型的选项
            showText(ts.getText());
        }
        if (selection instanceof IMarkSelection) {
            IMarkSelection ms = (IMarkSelection) selection;
            try {
                //显示 IMarkSelection 类型的选项
                showText(ms.getDocument().get(ms.getOffset(), ms.getLength()));
            } catch (BadLocationException ble) { }
        }
    }

    private void showItems(Object[] items) {
        tableviewer.setInput(items);
        pagebook.showPage(tableviewer.getControl());
    }

    private void showText(String text) {
        textviewer.setDocument(new Document(text));
        pagebook.showPage(textviewer.getControl());
    }
```

```
    public void createPartControl(Composite parent) {
        //创建 PageBook
        pagebook = new PageBook(parent, SWT.NONE);

        //创建 TableViewer
        tableviewer = new TableViewer(pagebook, SWT.NONE);
        tableviewer.setLabelProvider(new WorkbenchLabelProvider());
        tableviewer.setContentProvider(new ArrayContentProvider());

        //设置 tableviewer 为选择事件的提供者
        getSite().setSelectionProvider(tableviewer);

        //创建 textviewer
        textviewer = new TextViewer(pagebook, SWT.H_SCROLL | SWT.V_SCROLL);
        textviewer.setEditable(false);

        //添加选择事件监听器
        getSite().getWorkbenchWindow().getSelectionService().
        addSelectionListener(listener);
    }

    public void setFocus() {
        pagebook.setFocus();
    }

    public void dispose() {
        //删除选择事件监听器
        getSite().getWorkbenchWindow().getSelectionService().
            removeSelectionListener(listener);
        super.dispose();
    }

}
```

上例通过“setSelectionProvider(tableviewer)”代码把当前视图的 tableviewer 组件添加为选择事件的提供者，通过“addSelectionListener(listener)”代码添加了选择事件的监听器，当在 Eclipse 中选择了其他视图和编辑器的相关组件时，此选择监听器将会获得选择的组件信息，如图 24-4 所示。

通过选择树中的节点，在选择监听视图中显示相关的信息。用户还可以尝试选择文本和选择标记，在监听器视图中将会显示不同的信息。

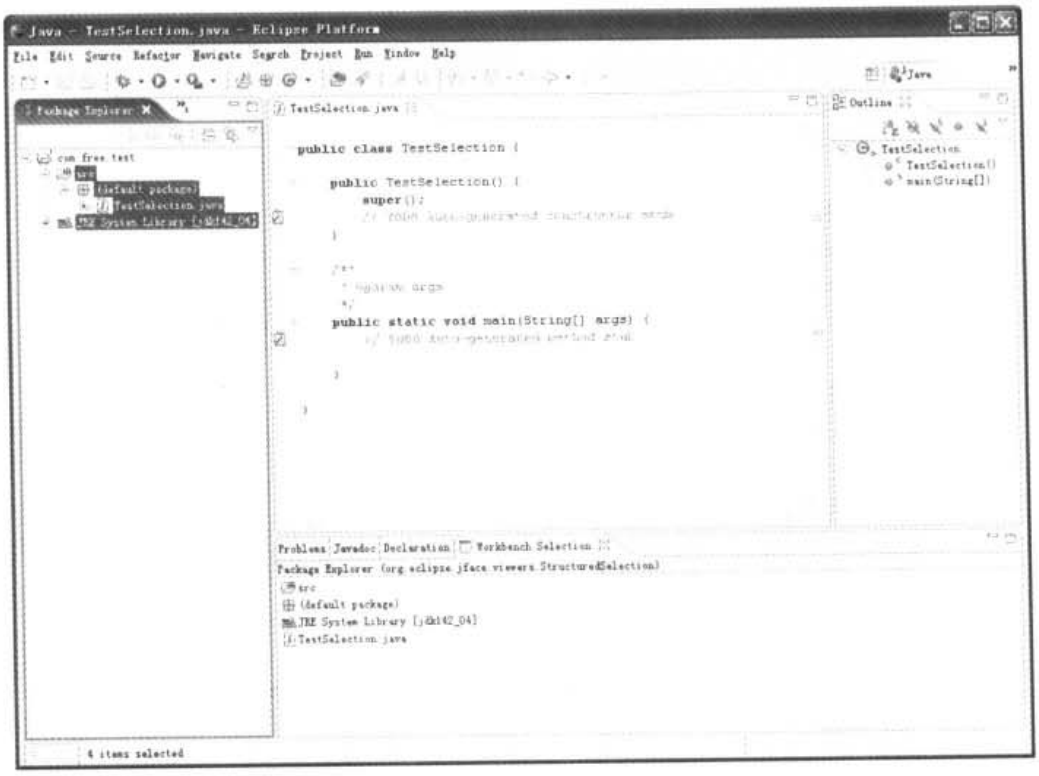


图 24-4　选择监听器

24.2　命 令 堆 栈

在“GEF 原理及实现”一章介绍了命令、命令执行、撤销和重复的操作，还简要地介绍了 GEF 框架通过命令堆栈维护命令执行、撤销和重复。

在 Eclipse 中，也大量地采用 Command、CommandManager 和 ICommandListener 等机制来控制命令的执行及监听。本节将介绍 GEF 命令堆栈（CommandStack）的运行方式，有兴趣的读者可以跟踪 Eclipse 源代码，查找其他框架命令的执行和撤销。

24.2.1　命令堆栈（CommandStack）

GEF 框架会维护 CommandStack，通过 CommandStack 来执行、撤销和重复命令的操作。CommandStack 中有两个堆栈（Stack）类型的数据——undoable 和 redoable：通过 undoable，CommandStack 知道有哪些命令可以撤销；通过 redoable，CommandStack 知道有哪些命令可能被重复。

当用户执行撤销操作时（例如按快捷键【Ctrl+Z】），GEF 框架会调用 CommandStack 的 undo 方法，执行撤销的操作。当用户执行重复的操作时（例如按快捷键【Ctrl+Y】），GEF 框架会调用 CommandStack 的 redo 方法，执行重复的操作。undo 和 redo 方法的代码片断如以下代码如示。

```
public void undo() {
    Command command = (Command)undoable.pop();
    notifyListeners(command, PRE_UNDO);
```

```
        try {
            command.undo();
            redoable.push(command);
            notifyListeners();
        } finally {
            notifyListeners(command, POST_UNDO);
        }
    }
    public void redo() {
        if (!canRedo())
            return;
        Command command = (Command)redoable.pop();
        notifyListeners(command, PRE_REDO);
        try {
            command.redo();
            undoable.push(command);
            notifyListeners();
        } finally {
            notifyListeners(command, POST_REDO);
        }
    }
```

undo 和 redo 方法会从相应堆栈的顶部取出命令，执行命令的 undo 和 redo 操作，然后把取出的命令放入另外一个堆栈中。通过这种机制，GEF 框架能够重复和撤销用户创建的命令，从而完成重复和撤销的操作。

CommandStack 不但完成重复和撤销命令的操作，还会协助完成命令的执行，当 GEF 框架要执行命令时，会调用 CommandStack 的 execute 方法执行相关命令，如以下代码片段所示。

```
    public void execute(Command command) {
        if (command == null || !command.canExecute())
            return;
        flushRedo();
        notifyListeners(command, PRE_EXECUTE);
        try {
            command.execute();
            if (getUndoLimit() > 0) {
                while (undoable.size() >= getUndoLimit()) {
                    ((Command)undoable.remove(0)).dispose();
                    if (saveLocation > -1)
                        saveLocation--;
                }
            }
            if (saveLocation > undoable.size())
                saveLocation = -1;
            undoable.push(command);
            notifyListeners();
        } finally {
            notifyListeners(command, POST_EXECUTE);
        }
    }
```

CommandStack 会调用 command 的 execute 方法执行命令，然后把 command 放入撤销命令的堆栈中。

另外，读者可以看到，在 execute、undo 和 redo 方法中都会调用 notifyListeners 方法，CommandStack 通过 notifyListeners 方法通知所有注册的监听器，命令堆栈已经发生了改变。

24.2.2 监听器

在 GEF 编辑器中，用户可以通过“getEditDomain().getCommandStack()”代码获得编辑器的命令堆栈，在 CommandStack 中可以通过 addCommandStackListener 方法添加命令堆栈的监听器，监听器是 CommandStackListener 类型的对象。CommandStackListener 接口定义如例程 24-5 所示。

例程 24-5 CommandStackListener.java

```
package org.eclipse.gef.commands;

import java.util.EventObject;

public interface CommandStackListener {

void commandStackChanged(EventObject event);

}
```

CommandStackListener 接口中声明了一个方法 commandStackChanged，当命令堆栈发生改变时，commandStackChanged 方法将会执行，如以下代码片段所示。

```
public void commandStackChanged(EventObject event)
{
    updateActions(stackActionIDs);
    setDirty(getCommandStack().isDirty());
}
protected void setDirty(boolean dirty)
{
    if (isDirty != dirty)
    {
        isDirty = dirty;
        //通知Eclipse，当前编辑器已经被修改
        firePropertyChange(IEditorPart.PROP_DIRTY);
    }
}
```

如以上代码所示，当命令堆栈改变时，编辑器将更新相应的 Action，另外通过 firePropertyChange 方法通知编辑器，当前编辑的内容已经被修改。

24.3 适 配 器

Eclipse 是一个可扩展的平台，但它也无法预料其中的对象今后可能需要它提供哪些服务。随着 Eclipse 的发展，人们会不断给它添加新的功能，而这些新的功能可能要求现有的类提供

新的服务或行为。对于 Eclipse 框架的开发者来说，保持 API 的稳定性是非常重要的，所以不能修改现有类的接口，也不能给现有的类添加新的接口。因此，有必要提供一种机制，能够修改 Eclipse 的接口，而又不改变基本接口的定义，这就是适配器将要完成的任务。

24.3.1 适配器接口（IAdaptable）

如果说适配器是 Eclipse 中核心的核心，这也一点都不为过，Eclipse 中到处都可以看到适配器的身影。Eclipse 中提供了 IAdaptable 接口， IAdaptable 接口的子类提供了把自己适配成其他接口不兼容对象的能力。IAdaptable 接口定义如例程 24-6 所示。

例程 24-6 IAdaptable.java

```
package org.eclipse.core.runtime;

public interface IAdaptable {

public Object getAdapter(Class adapter);
}
```

IAdaptable 接口非常简单，只有 getAdapter 方法。通过实现 IAdaptable 接口的 getAdapter 方法，对象可以把自己适配成接口不相容的其他对象。

getAdapter 方法只有一个 Class 类型的参数，当其他对象询问 IAdaptable 类型的对象是否支持某种类型的接口时，如果 IAdaptable 类型的对象支持此种类型的接口，则会返回一个此类型的变量，否则返回 null。Eclipse 中很多核心的类和接口都直接或间接地继承了 IAdaptable 接口，如图 24-5 所示。



图 24-5 IAdaptable 接口关系图

通过适配器的机制，用户可以把不相容的接口适配成其他的接口，例如，用户可以把编辑器中的内容通过大纲视图显示，可以把树节点的属性通过属性视图显示等，这些都是不相容接口类型的对象。

24.3.2 适配器的实现

当 Eclipse 平台或其他对象询问 IAdaptable 类型的对象是否支持某种类型的接口时，将会调用 getAdapter 方法，如以下代码片段所示。

```
protected IPropertySource getPropertySource(Object object) {
   if (sources.containsKey(object))
      return (IPropertySource) sources.get(object);

   IPropertySource result = null;
   if (propertySourceProvider != null)
      result = propertySourceProvider.getPropertySource(object);
   else if (object instanceof IPropertySource)
      result = (IPropertySource) object;
   //如果对象为 IAdaptable 类型，则通过 getAdapter 返回适配的对象
   else if (object instanceof IAdaptable)
      result = (IPropertySource) ((IAdaptable) object)
            .getAdapter(IPropertySource.class);

   sources.put(object, result);
   return result;
}
```

如上例为 Eclipse 源代码 PropertySheetEntry 类中的一个方法，当要获得属性的对象为 IAdaptable 类型时，将会通过对象的 getAdapter 方法询问该对象是否支持 IPropertySource 的接口，并返回此类型的变量或 null。

如果要支持某种不相容接口的类型，则实现了 IAdaptable 接口的对象要实现 getAdapter 方法返回相应的类型，如以下代码片段所示。

```
public Object getAdapter(Class adapter)
{
    //把当前对象适配为 GraphicalViewer 对象
    if (adapter == GraphicalViewer.class || adapter == EditPartViewer.class)
        return getGraphicalViewer();
    //把当前对象适配为 CommandStack 对象
    else if (adapter == CommandStack.class)
        return getCommandStack();
    //把当前对象适配为 EditDomain 对象
    else if (adapter == EditDomain.class)
        return getEditDomain();
    //把当前对象适配为 ActionRegistry 对象
    else if (adapter == ActionRegistry.class)
        return getActionRegistry();
    //把当前对象适配为 IPropertySheetPage 对象
    else if (adapter == IPropertySheetPage.class)
```

```
            return getPropertySheetPage();
        //把当前对象适配为 IContentOutlinePage 对象
        else if (adapter == IContentOutlinePage.class)
            return getOverviewOutlinePage();

        return super.getAdapter(adapter);
    }
```

上例所示为一个 GEF 编辑器适配方法的实现。通过上例可知，编辑器能适配为 CommandStack、IPropertySheetPage 和 IContentOutlinePage 等类型的接口（即可以通过不同的形式表现编辑器，而编辑器不用知道是如何实现的）。

IAdaptable 接口虽然只有定义一个方法，但它的使用是非常灵活的，另外，Eclipse 中还通过 IAdapterManager 和 IAdapterFactory 来共同完成适配器的实现。

24.4 UI 持久化

使用过 Eclipse 的读者都知道，当重新启动 Eclipse 时，Eclipse 会恢复上一次用户退出时的 UI 界面，Eclipse 通过 UI 持久化的机制实现界面的持久化。

24.4.1 持久化机制

Eclipse 中 IMemento 接口定义了对象持久化方式，通过 IMemento 接口的实现类，用户能把对象持久化到文件系统中。IMemento 接口定义如例程 24-7 所示。

例程 24-7 IMemento.java

```
package org.eclipse.ui;

public interface IMemento {
   public static final String TAG_ID = "IMemento.internal.id";

   public IMemento createChild(String type);

   public IMemento createChild(String type, String id);

   public IMemento getChild(String type);

   public IMemento[] getChildren(String type);

   public Float getFloat(String key);

   public String getID();

   public Integer getInteger(String key);

   public String getString(String key);
```

```
    public String getTextData();

    public void putFloat(String key, float value);

    public void putInteger(String key, int value);

    public void putMemento(IMemento memento);

    public void putString(String key, String value);

    public void putTextData(String data);
}
```

Eclipse 中 XMLMemento 实现了 IMemento 接口，通过 XMLMemento 对象，Eclipse 能把当前 UI 的状态持久化到文件系统中，其中 createChild 方法为创建下一级 IMemento 节点，getChild 方法返回下一级 IMemento 节点。

24.4.2 UI 持久化与恢复的实现

Eclipse UI 的设计目标之一，就是在多次会话之间保持用户体验的连贯性。当一次会话结束时，工作台会保存当前的配置和工作台窗口的内容；当用户开始新的一次会话时，工作台将这些状态恢复到上一次会话结束之前的样子。

Eclipse UI 持久化的过程比较复杂，因 UI 要持久化的内容比较多，包括编辑器、视图、视角、page 和 window 的状态，但持久化的方式一样。持久化的步骤如下。

（1）创建 XMLMemento 对象，并保存，实现类为 Workbench，如以下代码片段所示。

```
XMLMemento memento = XMLMemento
        .createWriteRoot(IWorkbenchConstants.TAG_WORKBENCH);
IStatus status = saveState(memento);o);
```

（2）得到所有的工作台窗口（WorkbenchWindow），保存所有窗口的状态，实现类为 Workbench，如以下代码片段所示。

```
IWorkbenchWindow[] windows = getWorkbenchWindows();
for (int nX = 0; nX < windows.length; nX++) {
    WorkbenchWindow window = (WorkbenchWindow) windows[nX];
    IMemento childMem = memento
            .createChild(IWorkbenchConstants.TAG_WINDOW);
    result.merge(window.saveState(childMem));
}
```

（3）得到工作台窗口包括的 Page（WorkbenchPage）页，并保存 Page 页的状态，实现类为 WorkbenchWindow，如以下代码片段所示。

```
Iterator itr = pageList.iterator();
while (itr.hasNext()) {
   WorkbenchPage page = (WorkbenchPage) itr.next();

   IMemento pageMem = memento
```

```
            .createChild(IWorkbenchConstants.TAG_PAGE);
        pageMem.putString(IWorkbenchConstants.TAG_LABEL, page.getLabel());
        result.add(page.saveState(pageMem));

        if (page == getActiveWorkbenchPage()) {
            pageMem.putString(IWorkbenchConstants.TAG_FOCUS, "true");
            }

        IAdaptable input = page.getInput();
        if (input != null) {
            IPersistableElement persistable = (IPersistableElement) input
                    .getAdapter(IPersistableElement.class);
            if (persistable == null) {
                WorkbenchPlugin
                        .log("Unable to save page input: "
                                + input
                                + ", because it does not adapt to IPersistableElement");
            } else {
                IMemento inputMem = pageMem
                        .createChild(IWorkbenchConstants.TAG_INPUT);
                inputMem.putString(IWorkbenchConstants.TAG_FACTORY_ID,
                        persistable.getFactoryId());
                persistable.saveState(inputMem);
            }
        }
    }
```

（4）保存 Page 页中所有的编辑器、视图、视角的状态，通过 EditorManager、ViewFactory 和 Perspective 保存编辑器、视图、视角的状态，实现类为 WorkbenchPage，如以下代码片段所示。

```
    IMemento childMem = memento
            .createChild(IWorkbenchConstants.TAG_EDITORS);
    result.merge(editorMgr.saveState(childMem));

    childMem = memento.createChild(IWorkbenchConstants.TAG_VIEWS);
    result.merge(getViewFactory().saveState(childMem));

    Iterator itr = perspList.iterator();
    while (itr.hasNext()) {
        Perspective persp = (Perspective) itr.next();
        IMemento gChildMem = childMem
                .createChild(IWorkbenchConstants.TAG_PERSPECTIVE);
        result.merge(persp.saveState(gChildMem));
    }
```

（5）把底层的 XMLMemento 保存到文件，实现类为 Workbench，如以下代码片段所示。

```
    private boolean saveMementoToFile(XMLMemento memento) {
        File stateFile = getWorkbenchStateFile();
        if (stateFile == null)
            return false;
        try {
            FileOutputStream stream = new FileOutputStream(stateFile);
```

```
            OutputStreamWriter writer = new OutputStreamWriter(stream, "utf-8");
            memento.save(writer);
            writer.close();
        } catch (IOException e) {
            stateFile.delete();
            MessageDialog.openError((Shell) null,
                    WorkbenchMessages.SavingProblem,
                    WorkbenchMessages.ProblemSavingState);
            return false;
        }

        return true;
    }
```

如上的步骤只是列出了持久化的过程，并没有完全列出持久化的所有代码，读者可以通过实现类找到具体代码的实现。

当 Eclipse 重新启动时，Eclipse 将通过从文件中读取相应的 XMLMemento 信息，恢复上一次退出 Eclipse 时的状态，具体步骤如下。

（1）从文件中读取 XMLMemento 信息，实现类为 Workbench，如以下代码片段如示。

```
final File stateFile = getWorkbenchStateFile();
if (stateFile == null || !stateFile.exists()) {
    String msg = WorkbenchMessages.Workbench_noStateToRestore;
    return new Status(IStatus.WARNING, WorkbenchPlugin.PI_WORKBENCH,
            IWorkbenchConfigurer.RESTORE_CODE_RESET, msg, null);
}
...
FileInputStream input = new FileInputStream(stateFile);
BufferedReader reader = new BufferedReader(
                    new InputStreamReader(input, "utf-8"));
IMemento memento = XMLMemento.createReadRoot(reader);
...
IStatus restoreResult = restoreState(memento);
```

（2）从 XMLMemento 读取所有窗口信息，并恢复窗口，实现类为 Workbench，如以下代码片段所示。

```
IMemento[] children = memento
   .getChildren(IWorkbenchConstants.TAG_WINDOW);

createdWindows = new WorkbenchWindow[children.length];

for (int i = 0; i < children.length; i++) {
    childMem = children[i];
    WorkbenchWindow newWindow = newWorkbenchWindow();
    createdWindows[i] = newWindow;
    newWindow.create();

    windowManager.add(newWindow);

    boolean restored = false;
```

```
        try {
            status.merge(newWindow.restoreState(childMem, null));
            try {
                newWindow.fireWindowRestored();
            } catch (WorkbenchException e) {
                status.add(e.getStatus());
            }
            restored = true;
        } finally {
            if (!restored) {
                createdWindows[i] = null;
                newWindow.close();
            }
        }
    }
```

（3）恢复所有 Page 的状态，实现类为 WorkbenchWindow，如以下代码片段所示。

```
WorkbenchPage newPage = null;
try {
   newPage = new WorkbenchPage(this, input);
   result.add(newPage.restoreState(pageMem, activeDescriptor));
   pageList.add(newPage);
   firePageOpened(newPage);
} catch (WorkbenchException e) {
   WorkbenchPlugin
         .log("Unable to restore perspective - constructor failed.", e);
   result.add(e.getStatus());
   continue;
}
```

（4）恢复所有编辑器、视图和视角的状态。通过 EditorManager、ViewFactory 和 Perspective 恢复编辑器、视图、视角的状态，实现类为 WorkbenchPage，如以下代码片段所示。

```
IMemento childMem = memento
      .getChild(IWorkbenchConstants.TAG_EDITORS);
result.merge(getEditorManager().restoreState(childMem));

childMem = memento.getChild(IWorkbenchConstants.TAG_VIEWS);
if (childMem != null)
   result.merge(getViewFactory().restoreState(childMem));
   …
IMemento perspMems[] = childMem
      .getChildren(IWorkbenchConstants.TAG_PERSPECTIVE);
Perspective activePerspective = null;
for (int i = 0; i < perspMems.length; i++) {
   try {
      Perspective persp = new Perspective(null, this);
      result.merge(persp.restoreState(perspMems[i]));
      IPerspectiveDescriptor desc = persp.getDesc();
      if (desc.equals(activeDescriptor))
         activePerspective = persp;
      else if ((activePerspective == null)
```

```
                && desc.getId().equals(activePerspectiveID))
            activePerspective = persp;
        perspList.add(persp);
        window.firePerspectiveOpened(this, desc);
    } catch (WorkbenchException e) {
    }
}
```

在 Eclipse 中，EditorManager 和 ViewFactory 负责编辑器和视图的持久化和恢复，另外还处理编辑器和视图的相关操作，读者可以跟踪 EditorManager 和 ViewFactory 中的方法，了解编辑器和视图持久化的实现。

24.5 深入 Workbench

通过上一节介绍 UI 持久化的实现，读者应该对 UI 持久化的过程有个大概的了解。本节将继续介绍 Workbench 中实现的几个核心类，旨在引导读者能够进一步去研究 Workbench 的实现。

24.5.1 Workbench 中实现类的关系

Workbench（工作台）定义了 Eclipse 的 UI 样式，包括编辑器、视图和视角。Workbench 建立在 SWT/JFace 之上。Workbench 中对象的关系如图 24-6 所示。

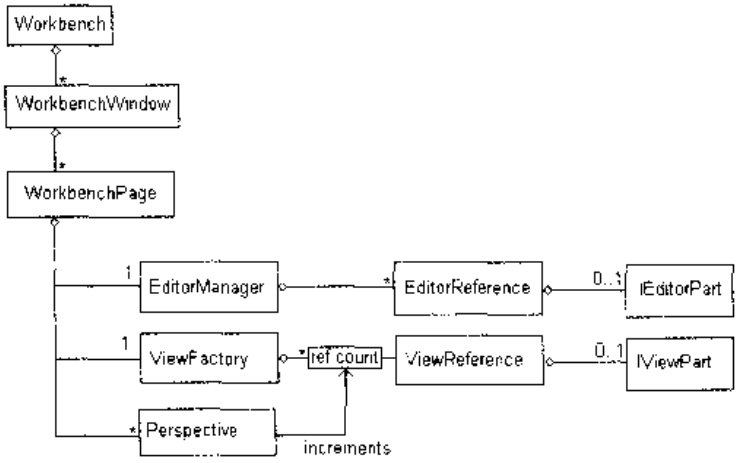


图 24-6 工作台对象之间的关系

Workbench 包含一个或多个 WorkbenchWindow，其中每一个 WorkbenchWindow 包含零或多个 WorkbenchPage。WorkbenchWindow 实现窗口，而 WorkbenchPage 实现窗口的内容，理论上说，WorkbenchWindow 能包含多个 WorkbenchPage（从上面持久化 Page 的代码也可以看出，Eclipse 支持一个窗口中有多个 Page），但实际上，在 WorkbenchWindow 中最多只有一个 WorkbenchPage 实现。

视图和编辑器是 WorkbenchPage 的一部分，WorkbenchPage 通过 ViewFactory 和 EditorManager 管理视图和编辑器。EditorManager 保存了编辑器的列表，编辑器共享资源，

ViewFactory 保存了视图引用的列表。

WorkbenchPage 包含视角的列表，视角又包含了视角定义的布局信息。尽管视角定义中包含了视图和编辑器区域的内容，但视角不会拥有视图和编辑器，它只包含了视图和编辑器的引用。工作台中对象所代表的区域如图 24-7 所示。

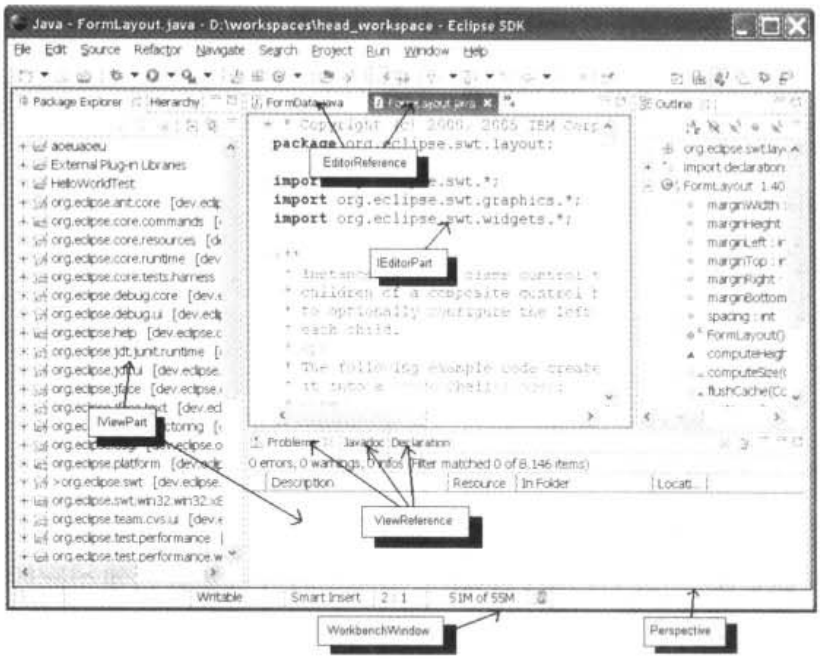


图 24-7 工作台对象

24.5.2 Part 的生命周期

在插件开发中，用户比较关心的是 Part 对象。用户通过 Reference 对象可以获得相应的 Part。Eclipse 中创建 Part 的开销是比较大的，通过 Reference 获得相应的 Part 的一个好处就是可以懒加载相应的 Part，通过这种方式，可以大大提高 Eclipse 的执行效率。

用户通常可以看到的是 IWorkbenchPartReference 对象、IViewReference 对象和 IEditorReference 对象，其中，WorkbenchPartReference 对象是最上层的 Part 对应的 Reference 对象的实现，如图 24-8 所示。

WorkbenchPartReference 会以懒加载的方式分配 SWT 资源，资源分配后，最后还要显示回收资源，WorkbenchPage 负责销毁 PartReference。

WorkbenchPartReference 中通过 state 状态位表示对应的 Part 的生命周期，状态值如下代码片断所示。

```
//表明 Part 还未创建
public static int STATE_LAZY = 0;
//表明 Part 正在创建
public static int STATE_CREATION_IN_PROGRESS = 1;
//表明 Part 已经被创建
public static int STATE_CREATED = 2;
//表明 Reference 已经不可用
public static int STATE_DISPOSED = 3;
```

通过 state 状态，WorkbenchPartReference 能管理 Part 的生命周期，如图 24-9 所示。

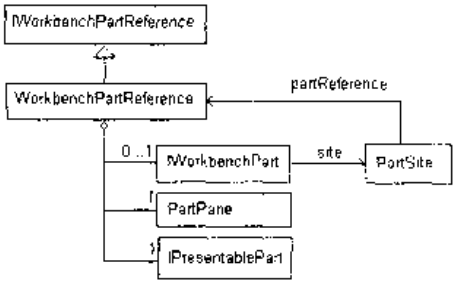


图 24-8　Part 对象关系

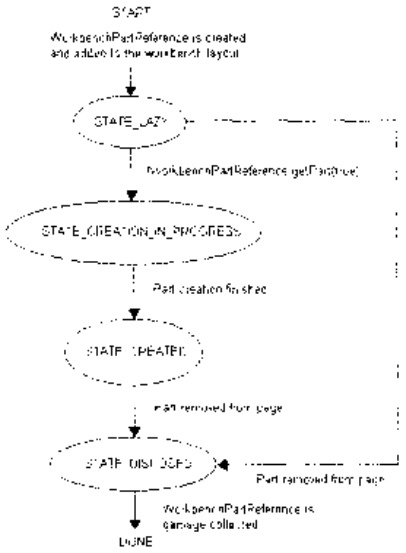


图 24-9　part 的生命周期

当 Reference 创建后，相应的 Part 是没有创建的，只有在 Reference 中调用 getPart，才会创建相应的 Part（Eclipse 中的宗旨是，只在实际使用的时候才会创建相应的资源，这样就不会因为一次性创建资源过多而影响系统的执行效率），如以下代码片段所示。

```
public final IWorkbenchPart getPart(boolean restore) {
   if (isDisposed()) {
      return null;
   }

   if (part == null && restore) {

      if (state == STATE_CREATION_IN_PROGRESS) {
         IStatus result = WorkbenchPlugin.getStatus(
               new PartInitException(NLS.bind(
                  "Warning: Detected recursive attempt by part {0} "+
                  "to create itself (this is probably, but not necessarily, a bug)",
                   getId())));
         WorkbenchPlugin.log(result);
         return null;
      }

      try {
         state = STATE_CREATION_IN_PROGRESS;

         IWorkbenchPart newPart = createPart();
         if (newPart != null) {
            part = newPart;
            getPane().getControl().addDisposeListener(prematureDisposeListener);
            part.addPropertyListener(propertyChangeListener);

            refreshFromPart();
```

```
                releaseReferences();

                fireInternalPropertyChange(INTERNAL_PROPERTY_OPENED);
            }
        } finally {
            state = STATE_CREATED;
        }
    }

    return part;
}
```

以上代码获得当前 WorkbenchPartReference 中对应的 Part。Eclipse 中 Part 都是通过相应的 Reference 得到的，另外，PartSite 能通过 getPartReference 得到相应的 IWorkbenchPartReference。

本节是笔者研究 Workbench 的一次探索，也希望读者能花更多的时间研究 Workbench 相关的源代码，通过学习 Eclipse 的源代码，相信读者能在此中受益匪浅。



要想单步跟踪 Eclipse 的源代码，可以用调试插件的方式，启动一个 Eclipse 工作台，在 Eclipse 源代码中的相应位置设置断点即可。

24.6 资源监听与访问

有很多的插件和工具都会关心 Workspace 中资源的改变，例如，当用户打开的编辑器所依赖的资源被删除掉时，应该先关闭打开的编辑器，还有，当 Workspace 中的资源被改变后，构建器要进行相应的构建等，这些都可以通过资源监听器实现。

24.6.1 资源监听器

Eclipse 插件实现资源监听的主要方式是实现资源监听器，通过资源监听器，插件就能够保证 Workspace 中的资源和模型同步。另外，资源监听器还能知道当前什么资源被改变，以及怎样被改变了。

资源监听器要实现 IResourceChangeListener 接口，并能通过 IWorkspace 的 addResourceChangeListener 方法注册。当不再需要资源监听器时，一定要调用 removeResourceChangeListener 方法把监听器从 IWorkspace 中删除。

IResourceChangeListener 接口的定义如例程 24-8 所示。

例程 24-8 IResourceChangeListener.java

```
package org.eclipse.core.resources;

import java.util.EventListener;

public interface IResourceChangeListener extends EventListener {

public void resourceChanged(IResourceChangeEvent event);
```

```
}
```

IResourceChangeListener 接口中只定义 resourceChanged 方法，当资源改变后，将会执行 IResourceChangeListener 实例的 resourceChanged 方法。实现 IResourceChangeListener 接口如下代码所示。

```
IResourceChangeListener listener = new IResourceChangeListener() {
  public void resourceChanged(IResourceChangeEvent event) {
    System.out.println("Something changed!");
  }
};
```

上例实现了一个简单的资源监听器，其中 IResourceChangeEvent 表示资源改变的事件的对象，通过 IResourceChangeEvent，用户获得资源改变的对象，例如“event.getDelta();”。实现资源监听器后，用户要负责把监听器注册到 Workspace 中，最后还应该删除注册的监听器，步骤如下。

（1）获得 Workspace 的实例，例如“IWorkspace workspace = ResourcesPlugin.getWorkspace();”。

（2）注册资源监听器，例如“workspace.addResourceChangeListener(listener);”。

（3）删除资源监听器，例如“workspace.removeResourceChangeListener(listener);”。

24.6.2 资源访问器

在 Workspace 中，资源是以树型结构组织的，用户可以通过树型结构访问被改变的资源树。通常，用户应该实现 IResourceDeltaVisitor 接口，以实现改变资源的访问。IResourceDeltaVisitor 接口定义如例程 24-9 所示。

例程 24-9 IResourceDeltaVisitor.java

```
package org.eclipse.core.resources;

import org.eclipse.core.runtime.CoreException;

public interface IResourceDeltaVisitor {

public boolean visit(IResourceDelta delta) throws CoreException;
}
```

IResourceDeltaVisitor 接口只定义 visit 方法，其中参数 IResourceDelta 表示 Workspace 中资源树的改变对象。IResourceDeltaVisitor 的实现类只要实现 visit 方法即可，如例程 24-10 所示。

例程 24-10 Visitor.java

```
class Visitor implements IResourceDeltaVisitor {
    public boolean visit(IResourceDelta delta) {
      switch (delta.getKind()) {
      case IResourceDelta.ADDED :
        break;
```

```
            case IResourceDelta.REMOVED :
                break;
            case IResourceDelta.CHANGED :
                break;
            }
        return true;
        }
    }
```

IResourceDelta 有一些整型的常量，指明了当前被改变资源的状态，例如“ADDED、NO_CHANGE、REMOVED”等，IResourceDelta有一个实现类ResourceDelta，通过ResourceDelta的accept方法，结合IResourceDeltaVisitor的实现类，共同实现了Visit设计模式。accept方法的实现如以下代码片段所示。

```
public void accept(IResourceDeltaVisitor visitor) throws CoreException {
    accept(visitor, 0);
}
public void accept(IResourceDeltaVisitor visitor, int memberFlags)
        throws CoreException {
    final boolean includePhantoms = (memberFlags & IContainer.INCLUDE_PHANTOMS) != 0;
    final boolean includeTeamPrivate =
        (memberFlags & IContainer.INCLUDE_TEAM_PRIVATE_MEMBERS) != 0;
    int mask = includePhantoms ? ALL_WITH_PHANTOMS : REMOVED | ADDED | CHANGED;
    if ((getKind() & mask) == 0)
        return;
    if (!visitor.visit(this))
        return;
    for (int i = 0; i < children.length; i++) {
        ResourceDelta childDelta = children[i];
        if (!includeTeamPrivate && childDelta.isTeamPrivate())
            continue;
        if (!includePhantoms && childDelta.isPhantom())
            continue;
        childDelta.accept(visitor, memberFlags);
    }
}
```

提示 如以上代码所示，ResourceDelta 和 IResourceDeltaVisitor 的实现类结合起来，实现了 Visitor 设计模式。Eclipse 中还使用了很多的设计模式，实现了 Eclipse 中很多核心的功能，有兴趣的读者可以先参考设计模式的实现，然后查找 Eclipse 中的实现，相信更有助于读者进一步理解设计模式的实现，而且对理解 Eclipse 核心的实现是非常有帮助的。

24.6.3 资源监听的实现

要实现资源监听器，并根据资源改变的内容处理资源改变事件，用户应该先实现 IResourceChangeListener 接口，并把监听器注册到 Workspace 中。另外，可以通过实现 IResourceDeltaVisitor 接口，访问修改的内容，如例程 24-11 所示。

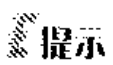
例程 24-11　DocIndexUpdater.java

```
public class DocIndexUpdater implements IResourceChangeListener {
  //显示列表
  private TableViewer table;
  private static final IPath DOC_PATH = new Path("MyProject/doc");
  //响应资源改变事件
  public void resourceChanged(IResourceChangeEvent event) {
    //处理 POST_CHANGE 事件
    if (event.getType() != IResourceChangeEvent.POST_CHANGE)
      return;
    //获得改变的对象
    IResourceDelta rootDelta = event.getDelta();
    IResourceDelta docDelta = rootDelta.findMember(DOC_PATH);
    if (docDelta == null)
      return;
    //创建改变的列表
    final ArrayList changed = new ArrayList();
    IResourceDeltaVisitor visitor = new IResourceDeltaVisitor() {
      public boolean visit(IResourceDelta delta) {
        //处理 IResourceDelta.CHANGED 事件
        if (delta.getKind() != IResourceDelta.CHANGED)
          return true;
        if ((delta.getFlags() & IResourceDelta.CONTENT) == 0)
          return true;
        IResource resource = delta.getResource();
        //只处理扩展名为 txt 的文件
        if (resource.getType() == IResource.FILE &&
            "txt".equalsIgnoreCase(resource.getFileExtension())) {
          //把改变的内容添加到列表
          changed.add(resource);
        }
        return true;
      }
    };
    try {
      //通过 Visitor 模式，遍历 docDelta
      docDelta.accept(visitor);
    } catch (CoreException e) {

    }
    if (changed.size() == 0)
      return;
    //显示在 table 中
    Display display = table.getControl().getDisplay();
    if (!display.isDisposed()) {
      display.asyncExec(new Runnable() {
        public void run() {
          //确认 table 是否存在
          if (table.getControl().isDisposed())
            return;
```

```
                table.update(changed.toArray(), null);
            }
        });
    }
}
}
```

如上代码实现了资源监听器，当资源为 txt 文件时，把文件显示在 table 中，用户可以通过监听器找到被改变的资源。



Eclipse 框架还支持很多其他监听器的实现，例如“IPartListener”，在此无法介绍全面，读者可以查找 Eclipse 源代码，一定能找到相关代码的实现。

24.7 小 结

本章希望带领读者了解 Eclipse 的一些实现内幕，但只凭本章的介绍是远远不够的，对于想掌握 Eclipse 开发的读者来说，还有很长的路要走。

本书只能作为学习 Eclipse 的一个切入点，引导读者如何开发 Eclipse 的插件。Eclipse 中最好的学习资料是它的源代码。通过本书的学习，读者应该有能力对 Eclipse 的源代码进行研究，相信读者能够在研究 Eclipse 源代码的过程中，越走越好。

附录 A　插件资源列表

Eclipse 是基于插件的体系架构，任何用户都可以提供插件。Eclipse 的成功和众多插件的开发成功是分不开的，在此列出一些常用的插件及简介，其中不乏成功的开源插件。用户在使用他人开发插件的同时，也可以研究这些插件的实现。

（1）Properties Editor

简介：Properties Editor 可以编辑 Java 的属性文件，并可以自动存盘为 Unicode 格式。

下载地址：http://propedit.sourceforge.jp/index_en.html

（2）Colorer Take

简介：Colorer Take 可以为上百种类型的文件按语法着色。

下载地址：http://colorer.sourceforge.net/

（3）XMLBuddy

简介：XMLBuddy 可以编辑 XML 文件。

下载地址：http://www.Xmlbuddy.com

（4）Code Folding

简介：Code Folding 可以加入多种代码折叠功能（比 Eclipse 自带的更多）。

下载地址：http://www.coffee-bytes.com/servlet/PlatformSupport

（5）Easy Explorer

简介：Easy Explorer 可以打开在 Eclipse 中选定文件所在的目录。

下载地址：http://easystruts.sourceforge.net/

（6）Fat Jar

简介：Fat Jar 可以打包插件，还可以方便地完成各种打包任务，能自动包含外部的包等。

下载地址：http://fjep.sourceforge.net/

（7）RegEx Test

简介：RegEx Test 可以测试正则表达式。

下载地址：http://brosinski.com/stephan/archives/000028.php

（8）Jigloo GUI Builder

简介：Jigloo GUI Builder 可以提供 Java 的 GUI 编辑插件。

下载地址：http://cloudgarden.com/jigloo/

（9）Profiler

简介：Profiler 是性能跟踪、测量工具，能跟踪、测量 BS 程序。

下载地址：http://sourceforge.net/projects/eclipsecolorer/

（10）AdvanQas

简介：AdvanQas 提供对 if...else 等条件语句的提示和快捷帮助（自动更改结构等）。

下载地址：http://eclipsecolorer.sourceforge.net/advanqas/

（11）Log4E

简介：Log4E 是 Log4j 插件，提供各种和 Log4j 相关的任务，如为方法、类添加一个 logger 等。

下载地址：http://log4e.jayefem.de/index.php/Main_Page

（12）VSSPlugin

简介：VSSPlugin 是 VSS 插件。

下载地址：http://sourceforge.net/projects/vssplugin

（13）Implementors

简介：当追踪方法代码时，Eclipse 默认是转到方法的接口类，而接口中只有方法名称没有具体的实现代码。此插件提供了追踪到具体实现类的实现代码的功能。

下载地址：http://eclipse-tools.sourceforge.net/implementors/

（14）Call Hierarchy

简介：Call Hierarchy 显示一个方法的调用层次。

下载地址：http://eclipse-tools.sourceforge.net/call-hierarchy/

（15）EclipseTidy

简介：EclipseTidy 基于 HTML Tidy 提供校验和格式化 HTML/XML 文件的功能。

下载地址：http://eclipsetidy.sourceforge.net/

（16）Checkclipse

简介：Checkclipse 可以检查代码的风格、写法是否符合规范。

下载地址：http://checkstyle.sourceforge.net/

（17）Hibernate Synchronizer

简介：Hibernate Synchronizer 是 Hibernate 插件，可以自动实现映射。

下载地址：http://www.binamics.com/hibernatesync/

（18）VeloEclipse

简介：VeloEclipse 是 Velocity 插件。

下载地址：http://propsorter.sourceforge.net/

（19）EditorList

简介：EditorList 能方便地列出所有打开的 Editor。

下载地址：http://editorlist.sourceforge.net/

（20）MemoryManager

简介：MemoryManager 可以实现内存占用率的监视。

下载地址：http://cloudgarden.com/memorymanager/

（21）TomcatPlugin

简介：TomcatPlugin 是支持 Tomcat 的插件。

下载地址：http://www.sysdeo.com/eclipse/tomcatPlugin.html

（22）Lomboz

简介：Lomboz 是 Eclipse 的一个 J2EE 插件，它将很多 Java 应用服务器、J2EE 组件和 Web 应用开发集成到 Eclipse 中，可以帮助 Java 开发者使用 Eclipse 建立、测试、部署 J2EE 应用。它支持 JSP 编写，包括语法着色和语法检查。

下载地址：http://lomboz.objectweb.org

（23）EclipseQuote

简介：EclipseQuote是为Eclipse IDE提供的插件，可以用于监视股市行情的走势。它可以在Eclipse视图中显示股票报价，可以增加新股票，更新价格，单击完成功能。

下载地址：http://sourceforge.net/projects/equo/

（24）JSEclipse

简介：JSEclipse可以编辑JavaScript文件，支持语法突出显示和包括了一些常用的面向对象的方法。支持的面向对象的功能有：构造函数、类方法、实例方法、类变量等。支持代码错误提示、警告报告等。

下载地址：http://www.interaktonline.com/Products/Eclipse/JSEclipse/Overview/?&error=1

（25）CSS Editor

简介：CSS Editor可以编辑复杂的CSS功能。

下载地址：http://sourceforge.net/projects/csseditor/

（26）eclipse UML

简介：UML建模插件功能其全。

下载地址：http://www.omondo.com/

（27）commonclipse

简介：commonclipse是为jakarta commons-lang用户提供的Eclipse插件。它可以使用commons-lang生成器自动生成下列方法：toString()、hashCode()、equals(Object)、compareTo(Object)。

下载地址：http://sourceforge.net/projects/commonclipse/

（28）AntView

简介：AntView是一个Elipse插件，可从Eclipse IDE中运行Ant文件。

下载地址：http://r2tech-eclipse.sourceforge.net/

（29）StrutsIDE

简介：StrutsIDE是Struts框架开发工具，它提供了struts-config.xml配置文件的图形管理界面。

下载地址：http://amateras.sourceforge.jp/cgi-bin/fswiki_en/wiki.cgi?page=StrutsIDE

（30）SQLExplorer

简介：SQLExplorer可以通过JDBC访问几乎任何一种数据库，同时也支持像Hibernate这样的工具访问数据库。

下载地址：http://sourceforge.net/projects/eclipsesql

（注：以上下载地址均为参考地址。）

附录 B　插件开发法则

《Contributing to Eclipse》一书介绍了 Eclipse 开发中的常见法则，这些法则是此书的精髓，在此列出这些开发法则，希望读者在开发中慢慢体会。

B.1　扩展者法则

贡献法则（Contribution Rule）：一切皆是贡献。

遵循法则（Conformance Rule）：插件必须遵循预期的接口。

共享法则（Sharing Rule）：增加，不要取代。

有样学样法则（Monkey see/Monkey Do Rule）：遇到问题时，首先复制类似插件的结构。

相关性法则（Relevance Rule）：只有在操作有可能成功时才显示你所贡献的操作。

整合法则（Intergration Rule）：要整合，不要分裂。

责任法则（Responsibility Rule）：明确指出你开发的插件是问题的源头。

针对 API 契约编程法则（Program To API Contract Rule）：首先检查 Eclipse API 契约，然后针对契约编程。

“其他”法则（Other Rule）：让用户选择所有东西，但把那些通常不用于当前世界的选项放在 Other 对话框中。

IResource 适配法则（Adapt To IResource Rule）：应该尽量为领域对象定义 IResource 适配器。

分层法则（Strata Rule）：将与语言无关的功能与特定于具体语言的功能分开，将核心功能与 UI 功能分开。

使用连贯性法则（User Continuity Rule）：在多次会话之间，应该保持用户界面状态一致。

B.2　促成者法则

邀请法则（Invitation Rule）：尽可能地邀请别人为你的作品做出贡献。

懒加载法则（Lazy Loading Rule）：只有在真正需要的时候才加载插件。

安全平台法则（Safe Platform Rule）：作为扩展点的提供者，你必须保护好自己，不要让扩展者的误操作给你造成损失。

公平竞赛法则（Fair Play Rule）：所有使用者遵守同样的游戏规则，包括我自己。

明确扩展法则（Explicit Extension Rule）：明确说明瓶盖的什么地方可供扩展。

发散性法则（Diversity Rule）：一个扩展点接纳多个扩展。

良好防御法则（Good Fences Rule）：如果要交出程序的控制权，首先保护好你自己。

用户决定法则（User Arbitration Rule）：如果有多个选择，由用户决定使用哪一个。

明确 API 法则（Explicit Rule）：将 API 与插件内部使用的类分开。

稳定性法则（Stability Rule）：如果你已经开始邀请其他人做出贡献，就不要再改变游戏规则。

保守 API 法则（Defensive API Rule）：只暴露你有信心的 API，但同时也应该做好准备暴露更多的 API，因为使用者会要求你这样做。

B.3 发布者法则

许可法则（License Rule）：每项贡献品都应该提供许可证。

附录 C　在 Eclipse 中使用 JUnit

在 Eclipse 中使用 JUnit 的具体介绍如下。

C.1　JUnit 简介

随着软件项目的逐渐增大，软件测试在软件开发中的地位显得越来越重要。如果软件项目没有良好的测试流程，随着系统的增大，无论项目管理人员还是软件开发人员都会对项目的前景失去信心，甚至会对项目的目标产生分歧，因为长期以来没有对程序代码和系统设计进行有效地控制，很多问题都被暂时掩盖或逐渐演化成其他的问题。软件开发周期越长，就会使得问题进化的版本越多，最后造成的结果是“剪不断，理还乱”。

单元测试是整个测试流程中最基础的部分，它们要求程序员尽可能早地发现问题，并给予控制，这是其一。另外，如果集成测试出现问题，它们可以帮助诊断。这样就为在软件开发流程中建立高效的事件反应机制打下了坚实基础。

JUnit 就是为 Java 程序开发者实现单元测试提供一种框架，使得 Java 单元测试更规范有效，并且更有利于测试的集成。

JUnit 共有 7 个包，核心的包就是 junit.framework 和 junit.runner。Framework 包负责整个测试对象的构架，Runner 负责测试驱动。

JUnit 有 4 个重要的类：TestSuite、TestCase、TestResult、TestRunner。前 3 个类属于 Framework 包，后 1 个类在不同的环境下是不同的。如果使用的是文本测试环境，使用的将是 junit.textui.TestRunner。各个类的职责如下。

（1）TestResult：负责收集 TestCase 所执行的结果。它将结果分为两类，客户可预测的 Failure 和没有预测的 Error，同时负责将测试结果转发到 TestListener（该接口由 TestRunner 继承）处理。

（2）TestRunner：客户对象调用的起点，负责对整个测试流程的跟踪。能够显示返回的测试结果，并且报告测试的进度。

（3）TestSuite：负责包装和运行所有的 TestCase。

（4）TestCase：客户测试类所要继承的类，负责测试时对客户类进行初始化，以及测试方法调用。

另外，还有两个重要的接口：Test 和 TestListener。

（1）Test：包含两个方法，run() 和 countTestCases()。它是对测试动作特征的提取。

（2）TestListener：包含 4 个方法，addError()、addFailure()、startTest()和 endTest()。它是对测试结果的处理及测试驱动过程动作特征的提取。

C.2 Eclipse 开发、调试测试程序

Eclipse 的 SDK 中已经提供了 JUnit 的测试插件，它已经把 JUnit.jar 加入了插件中，用户可以通过 Eclipse 直接创建自己的测试程序，步骤如下。

（1）创建 Java 项目。

（2）选择“File”→“New”→“other”菜单，弹出新建对话框，选择要创建的测试单元，如图 C-1 所示。



图 C-1 选择 JUnit 向导

在此对话框中可以选择 TestCase 或者 TestSuite 创建用户的测试单元。

（3）单击“Next”按钮，在创建 TestCase 的对话框中输入增加测试的相关选项，如图 C-2 所示。

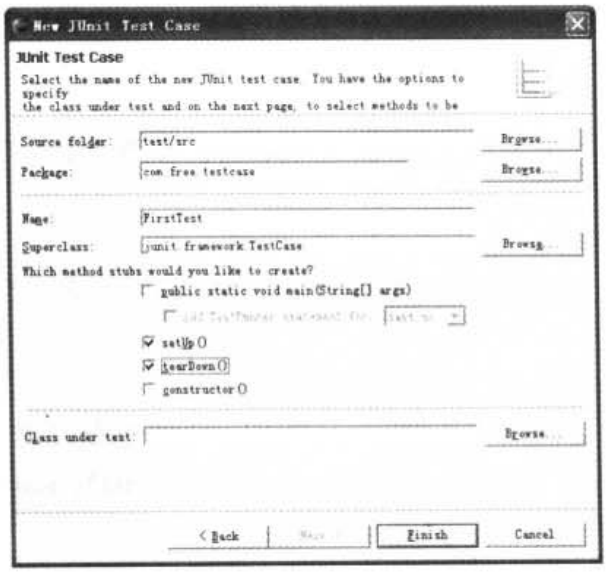


图 C-2 创建测试类

在创建测试类的对话框中能选择是否生成 setUp 和 tearDown 的方法等参数，在创建的类中，用户可以增加自己的测试方法，代码如例程 C-1 所示。

例程 C-1　FirstTest.java

```
package com.free.testcase;

import junit.framework.TestCase;

public class FirstTest extends TestCase {

    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testAdd() {
        assertTrue(6 == 6);
    }

    public void testEquals() {
        assertEquals(12, 12);
    }

}
```

这是一段很简单的代码，目的在于演示如何在 Eclipse 中加入单元测试，JUnit 会执行 test 开头的测试方法。

执行单元测试很简单，选择文件的右键菜单“Run as”→“JUnit Test”，将会执行所选的单元测试。上例的测试结果是正确的。在 Eclipse 中会通过 JUnit 视图显示测试结果，如图 C-3 所示。

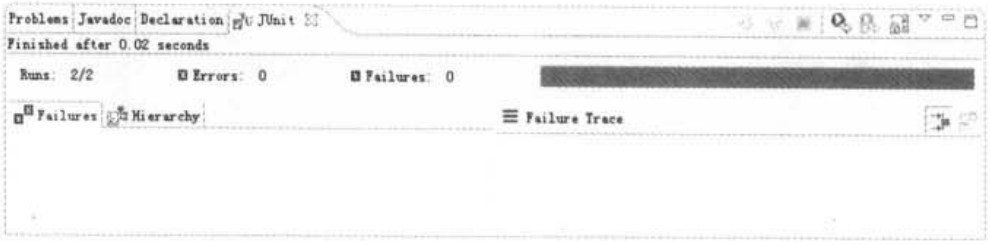


图 C-3　单元测试结果

附录 D　Visual Editor[①]

与许多 Eclipse.org 项目类似，Visual Editor（简称 VE）项目的目标是构建一个用于构建工具的工具。Visual Editor 是用于构建 AWT/Swing 及 SWT 应用程序的 GUI 构建器，这是一个期待已久的 Eclipse 特性。在 VE 1.0 版本中，已经增加了对于 SWT 的支持。在本文中，用户将获得关于 Visual Editor 及其背后技术的概览，以及 Visual Editor 用于构建 AWT/Swing 和 SWT 应用程序特性的简短示范。

D.1　Eclipse Visual Editor 项目介绍

仔细阅读关于 Eclipse 及其竞争者的优缺点的任何讨论线索，用户将会发现关于各种特性的提示信息，这些特性在其中各方互有优劣，或者完全缺乏。直到最近，对于 Eclipse，似乎经常被提起的事情是它缺少 GUI 构建器：一个用于构建图形用户接口的图形化工具。值得高兴的是，通过 2003 年 11 月发起的 Eclipse Visual Editor 项目及随后 Visual Editor 0.5 的迅速发布，这种情况已经得到弥补。Visual Editor 0.5 允许用户通过一个完全的 WYSIWYG（所见即所得）图形化编辑器来创建 AWT/Swing 应用程序。

与 Eclipse 自身类似，Visual Editor 也基于 IBM 的代码贡献。熟悉 Websphere Studio Application Developer 5.x 中的 Visual Editor 的开发人员将会发现 Eclipse 的 Visual Editor 几乎与之完全相同。

与 Eclipse.org 组织下的许多其他项目（包括 Eclipse）类似，Visual Editor 项目所确定的目标是非常宏伟的：构建一个用于构建图形化编辑工具的工具。尽管最初发行版本中对于 AWT/Swing 的支持已经完成，但是 Visual Editor 的计划远不止于此。技术基础正在进行重新设计，从而使得它对于编程语言和所支持的图形工具集来说是不相关的。在现有版本中，Visual Editor 已经实现了对 SWT 的支持。另外，可能还会提供潜在的除了 Java 语言以外的其他语言的实现（例如 C++）。

D.2　Visual Editor 的内幕

Visual Editor 的第一个具体实现，作为一个针对 AWT/Swing 的 GUI 构建器，对于 GUI 开发人员来说已经是足够令人满意的了，但是如果用户是那种希望了解内幕的开发人员，那么还可以看到许多东西：Visual Editor 利用一些非常有趣的技术，这些技术本身都是有用的。如

① 本文引自 IBM 中文网站的“利用 Eclipse Visual Editor 项目构建 GUI”一文，地址为“http://www-128.ibm.com/developerworks/cn/linux/opensource/os-ecvisual/index.html”，但由于此文章写得比较早，其中很多特性及操作方式已经发生了改变，笔者对此文进行了相关的修改，另外添加了开发 SWT 组件的内容。

果用户对于构建自己的图形化编辑器或者对工具建模感兴趣的话，那么现有的 Visual Editor 实现对于用户可以完成的事物来说只是一个线索。

Visual Editor 所利用的最为明显的工具是 GEF，即图形化编辑框架（Graphical Editing Framework）。GEF 建立于本地 Eclipse 图形化工具集 SWT 之上，以使得开发一个图形化编辑器或者所见即所得文本编辑器更为容易。如果用户熟悉 SWT（或者 AWT/Swing，在这方面它们是类似的）中的图形原语，那么用户会知道绘制和处理任意的形状（例如矩形、箭头和椭圆）是比较困难的，更不用说管理它们之间的关系以及它们所代表的数据模型了。

GEF 被划分为两个部分：第一部分是 Draw2d 插件，这是一个轻量级的绘图和呈现包，用于帮助用户绘制图形；第二部分是 GEF 插件，除了其他工具以外，这一部分中还增加了选择和创建工具、工具选项板，以及用于在数据模型和视图之间进行映射的控制器框架。

GEF 是一个模型无关的框架，但是作为 Visual Editor（及其他生成代码的图形化工具）的一部分，它在后台使用 Eclipse 建模框架（Eclipse Modeling Framework，EMF），以及在模型、Java 类和图形化表示之间进行映射，其中模型是使用 XML 元数据交换（XML Metadata Interchange，XMI）在内部存储的。EMF 的重要特性之一是它确保所有这些映射都是一对一的；所以尽管 XMI 可以被认为是模型的标准表示，但是在代码和图形之间来回切换并不会丢失任何信息。这就是为什么 Visual Editor 只需要保存模型的一种表示（即 Java 源代码），以及开发人员可以自由地在图形化编辑器之外编辑该源代码的原因。

D.3 利用 Visual Editor 开发 AWT/Swing 应用程序

正如前面所提到的，目前可用的版本 1.x 是一个完全的 AWT/Swing GUI 构建器。它与 Eclipse 3.x 共同工作，并且与其他 IDE 中的 GUI 构建器相比毫不逊色。首先，它生成高质量的代码，可以与一个有经验的 GUI 开发人员通过手工开发的代码相媲美，并且不带有使得修改变得困难的特殊工件（artifact）。其次，它强大的分析能力允许完全的代码来回转换，因此对源代码的修改几乎立即反映在图形化编辑器中。

在手工构建 Swing 应用程序时，最为乏味的任务之一就是使用布局管理器来管理组件的位置。因为 Visual Editor 是一个所见即所得图形化编辑器，所以利用它很容易在用户接口中获得用户想要的外观和行为。而且，因为它可以在不同的布局管理器之间自动映射，所以用户可以使用一个 null 布局来创建用户的应用程序外观，然后切换到一个栅格单元（grid-bag）布局。使用 null 布局能够容易地准确获取用户想要的布局，栅格单元布局则允许布局在窗口尺寸改变时能够运行良好。

在下面几节中，我们将快速地浏览 Visual Editor 及它的一些最有趣的特性。如果用户想随同实践的话，需要安装 Eclipse 3.x 和 Visual Editor 1.x，而且，Visual Editor 还需要另外两个插件，即 EMF 和 GEF。

D.4 Visual Editor 工具

在安装完 Visual Editor 之后，下次创建新的 Java 项目时，用户将会发现一些新的特性。假设用户已经创建了一个名为 VEPExample 的项目。在 Visual Editor 中用户可以选择 Visual

Class 创建 AWT/Swing 和 SWT 的应用。可以通过“New”菜单项选择相关选项，输入创建的类名即可，如图 D-1 所示。

选择完之后，单击“Finish”按钮创建可视类并使用 Visual Editor 打开它。用户将会注意到，与常规的 Java 编辑器不同，Visual Editor 具有三个不同的部分。顶部是一个图形化编辑器，其中显示用户的可视类在运行时的可能形状；右边是窗口部件的一个列表，这些窗口部件可以被拖放到用户的应用程序中；底部是源代码（见图 D-2）。



图 D-1 创建新的可视类



图 D-2 利用 Visual Editor 编辑 Swing 类

通过向下滚动源代码并找到 initialize()方法，用户可以看到源代码和图形化视图之间的交互。initialize()方法用于设置应用程序窗口的初始尺寸。

```
private void initialize() {
    this.setSize(300, 200);
    this.setContentPane(getJContentPane());
    this.setTitle("JFrame");
}
```

如果将第一个数字（即宽度）改变为一个新值，例如 600，用户将看到上面的图形化表示马上会改变宽度以反映这个新值。

除了 Visual Editor 之外，用户会注意到 Java 视角（透视图）中还有两个新的视图：位于左下部的 Java Beans 视图和位于右上部的 Properties 视图。用户可能知道，Swing 的设计特性之一是，每个组件，例如用户刚才创建的框架类及添加到它之上的任何其他窗口部件，都是一个 Java Bean。Java Beans 视图允许用户容易地在类中导航到这些组件。最初，在“this”（指当前在 Visual Editor 中的类）之下的唯一入口是 jContentPane。用户可能知道，并不直接向 JFrame 添加组件，而是将组件添加到它的内容面板。单击 jContentPane 将指向框架类中的 getJContentPane() 方法（见图 D-3）。

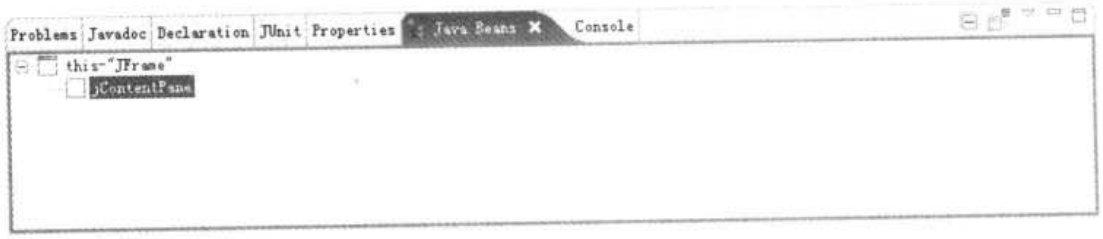


图 D-3　Java Beans 视图

Visual Editor 所增加的另一个视图是 Properties 视图，它显示一个 Java Bean 的属性。在这里，例如在 JavaBeans 视图中选择 jContentPane 之后，用户可以改变它所使用的布局管理器（在这样做之前，用户可能希望在 Editor 窗口中浏览一下源代码，查看它是通过调用带有一个 java.awt.BorderLayout 对象的 jContentPane.setLayout()方法来设置布局管理器的）。一些属性允许用户随意地输入文本，但是其他属性则提供一个更为合适的接口；针对布局管理的属性使用一个下拉列表，使用户只能在有效的布局管理器中进行选择。单击默认值 BorderLAyout，然后向上滚动所显示的列表并选择 null 布局管理器（见图 D-4）。

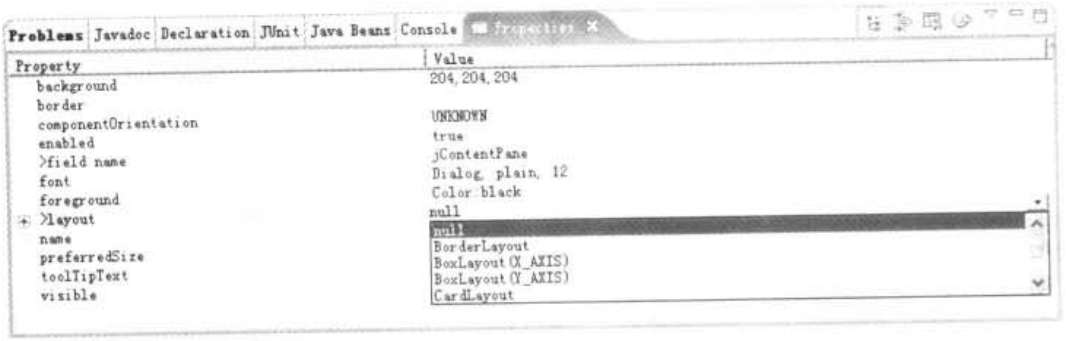


图 D-4　选择 null 布局管理器

在做出该改动之后，用户将看到在源代码中，jContentPane.setLayout() 现在通过一个 null 值被调用。如果用户希望自己证实 Properties 视图和 Editor 之间的交互在两个方向上都能够工作，用户可以尝试在源代码中将 null 改回为新的 java.awt.BorderLayout()，并且确认在 Properties 视图中该值自动改变。

D.5　创建和启用用户接口

一旦已经为用户的应用程序创建了一个框架，就可以开始添加窗口部件以允许用户与用户的应用程序进行交互。让我们添加一个复选框用于开关消息。首先，单击 JCheckbox 窗口部件，然后在图形化编辑器中的框架内单击。如果用户使用的是 null 布局管理器，用户会注意到可以将它放置在内容面板的任何地方；如果用户使用的是 BorderLayout，用户可以选择将它放置在北部、南部、东部、西部或者中部。

接着，单击 JLabel 窗口部件并在前面所添加的复选框中单击“Next”。使用 Properties 视图，将文本改为“Unchecked”，然后调整文本框的尺寸使得文本能够恰好适中。（另一个方法是，首先扩展该框然后单击左上角，将打开一个文本域，在这里用户可以键入文本。）

现在，用户的框架看起来应该类似于图 D-5 所示。

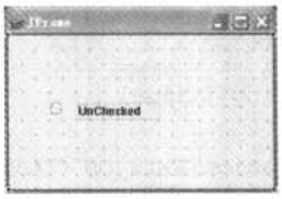


图 D-5 具有两个窗口部件的框架

如果用户现在开始运行该应用程序，当然不会发生太多的事情。用户可以将该复选框单击为打开或者关闭，因为 Swing 会为用户负责这件事情，但是要想让它实际做某些事情，用户还需要添加一些代码。如果熟悉 Swing 事件模型，用户就知道需要为复选框添加一个活动监听器，从而每当用户改变它时，监听器都可以执行某个动作。要使用 Visual Editor 添加一个监听器，可在图形化编辑器中用鼠标右键单击该复选框，并且从显示的上下文菜单中选择“Events”→“Action Performed”。这将把实现为匿名类的活动监听器的代码添加到复选框的初始化代码中。

```
private JCheckBox getJCheckBox() {
    if (jCheckBox == null) {
        jCheckBox = new JCheckBox();
        jCheckBox.setBounds(new java.awt.Rectangle(43,68,21,21));
        jCheckBox.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()");
            }
        });
    }
    return jCheckBox;
}
```

用户可以看到，在需要添加代码的地方已经利用一个 TODO 注释进行标识。让我们改动代码，以便每当复选框发生变化时，它旁边的标签也会发生变化以反映该复选框的状态。在用户做出改动之后，新的代码看起来应该是这样的。

```
jCheckBox.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(java.awt.event.ActionEvent e) {
    jLabel.setText(
      jCheckBox.isSelected() ? "Checked" : "Unchecked");
  }
});
```

D.6 运行 Visual Class

Visual Editor 可以让用户方便地启动 Java Bean，而不需要一个 main() 类。当用户正在测试一个组件（例如 JPanel），而该组件与最终将要包含它的应用程序相分离的时候，这会特别方便。要以这种方式启动用户创建的简单测试应用程序，请确保在 Editor 中已经选中 Test.java，然后从 Eclipse 主菜单中选择“Run”→“Run As”→“Java Bean”。

另一个方法是，由于这是一个 JFrame，用户还可以通过像下面这样完成 main() 方法，

来将 Test 作为一个 Java 应用程序运行。

```
public static void main(String[] args) {
    Test test = new Test();
    test.setDefaultCloseOperation(EXIT_ON_CLOSE);
    test.setVisible(true);
}
```

从主菜单中选择“Run”→“Run As”→“Java Application”来运行它。无论以哪种方式运行它，用户都应该看到，每当单击该复选框时，标签就会相应地发生变化。

D.7 Visual Editor 对 Eclipse 的贡献

Visual Editor 不但能对 AWT/Swing 提供完美的支持，还能支持 Eclipse 中的 SWT 组件可见即所得的方式创建，以及可以创建 Eclipse 中的 EditorPart 和 ViewPart 中 SWT 组件的构建。

Visual Editor 还没有提供对 JFace 的支持，但这已经足够了。用户可以通过 Visual Editor 中可见即所得的方式定制 SWT 组件，以及设定布局的方式，这对初学者来说，无疑是学习 SWT 的福音。

创建 SWT 的组件和创建 Swing 的组件类似，用户可以把组件直接拖动到 SWT 的设计器中。图 D-6 所示为用 SWT 设计器设计的简单组件。



图 D-6 Visual Editor 的 SWT 设计器

通过 SWT 设计器设计出的组件窗口，Visual Editor 将会自动同步生成 SWT 的代码，如例程 D-1 所示。

例程 D-1 SwtTest.java

```
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
```

```
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

public class SwtTest extends Composite {

    private Label label = null;
    private Label label1 = null;
    private Label label2 = null;
    private Text text = null;
    private Text text1 = null;
    private Text text2 = null;

    public static void main(String[] args) {
        Display display = Display.getDefault();
        Shell shell = new Shell(display);
        shell.setLayout(new FillLayout());
        shell.setSize(new Point(300, 200));
        SwtTest thisClass = new SwtTest(shell, SWT.NONE);
        shell.open();

        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }

    public SwtTest(Composite parent, int style) {
        super(parent, style);
        initialize();
    }

    private void initialize() {
        label = new Label(this, SWT.NONE);
        label.setBounds(new org.eclipse.swt.graphics.Rectangle(64,38,37,12));
        label.setText("姓名");
        label1 = new Label(this, SWT.NONE);
        label1.setBounds(new org.eclipse.swt.graphics.Rectangle(63,74,30,12));
        label1.setText("年龄");
        label2 = new Label(this, SWT.NONE);
        label2.setBounds(new org.eclipse.swt.graphics.Rectangle(64,111,53,12));
        label2.setText("出生年月");
        text = new Text(this, SWT.BORDER);
        text.setBounds(new org.eclipse.swt.graphics.Rectangle(141,34,70,18));
        text.setText("山树满");
        text1 = new Text(this, SWT.BORDER);
        text1.setBounds(new org.eclipse.swt.graphics.Rectangle(142,70,70,18));
        text1.setText("29");
        text2 = new Text(this, SWT.BORDER);
        text2.setBounds(new org.eclipse.swt.graphics.Rectangle(142,108,70,18));
        text2.setText("1977年4月");
```

```
            setSize(new Point(300, 200));
        }

    }
```

用户可通过运行 SWT 程序和 Java Bean 两种方式运行生成的代码，程序运行效果如图 D-7 所示。

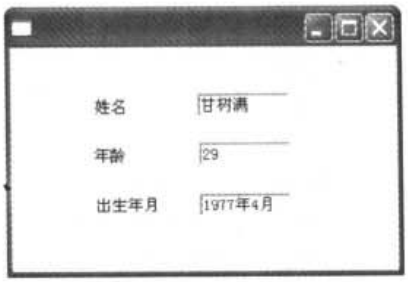


图 D-7　SWT 程序运行效果

通过 SWT 设计器不但可以通过可见即所得的方式设计组件，还能为组件添加相关的布局信息和组件的事件响应类。

添加组件响应事件的方式非常简单，用户可以通过组件的右键菜单完成此操作，选择右键菜单“Envents”→“Add Events”打开选择事件对话框，选择希望添加的事件，如图 D-8 所示。

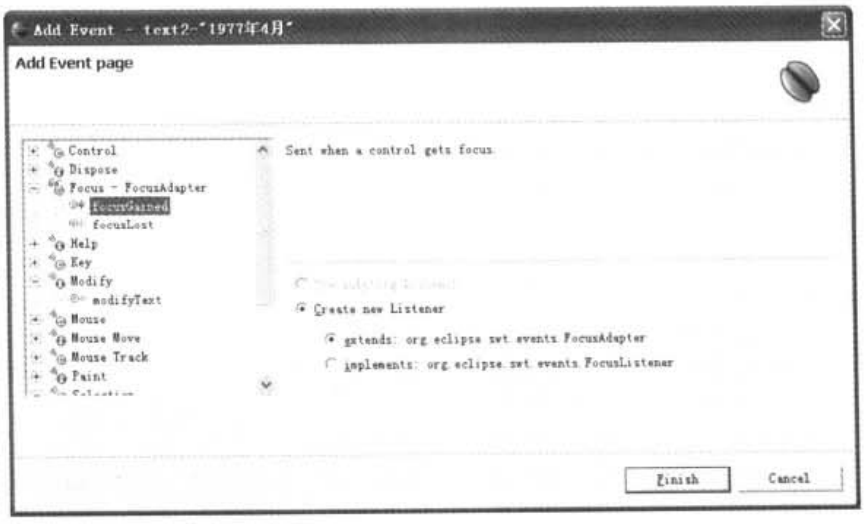


图 D-8　选择添加事件

通过选择添加事件的对话框，用户能为组件添加相关的响应事件。如图 D-8 所示，添加了焦点获得事件，Visual Editor 将会为该事件添加如下的事件响应代码。用户可以直接修改事件响应方法，添加用户对事件响应的动作。

```
        text2.addFocusListener(new org.eclipse.swt.events.FocusAdapter() {
            public void focusGained(org.eclipse.swt.events.FocusEvent e) {
                System.out.println("focusGained()");
            }
        });
```

除了可以添加组件的响应事件外，用户还能通过属性视图对所选组件添加布局信息。要能够合理地添加布局信息，用户应该对 SWT 的布局有所了解，然后在属性视图中选择相应的布局类和布局数据就可以了。

在 Visual Editor 中，用户还可以创建和 EditorPart 及 ViewPart 相关的 Java 类。通过设计器，用户可以快速开发相关的 Part。要创建相应的 Part，只需要在创建对话框中选择 RCP 节点中相应的 Part 即可，如图 D-9 所示。

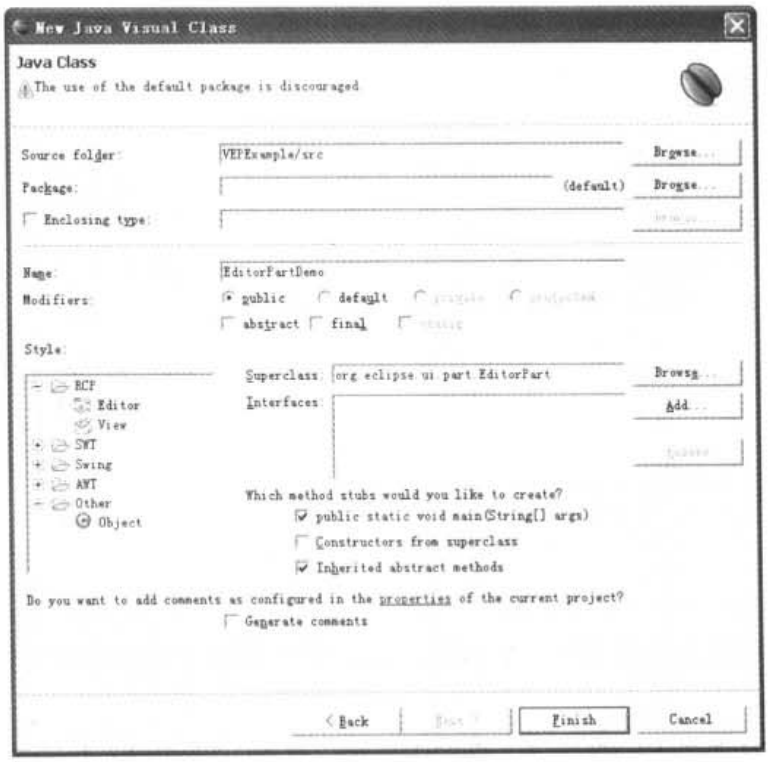


图 D-9　选择 Part 对话框

通过 Visual Editor 可以快速给出 Part 中相应的代码，如例程 D-2 所示。

例程 D-2　EditorPartDemo.java

```
import org.eclipse.swt.SWT;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IEditorSite;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.part.EditorPart;

public class EditorPartDemo extends EditorPart {

    public static final String ID = "EditorPartDemo";
    private Composite top = null;
    public void doSave(IProgressMonitor monitor) {

    }
```

```
    public void doSaveAs() {

    }

    public void init(IEditorSite site, IEditorInput input)
            throws PartInitException {

    }

    public boolean isDirty() {
        return false;
    }
    public boolean isSaveAsAllowed() {
        return false;
    }
    public void createPartControl(Composite parent) {
        top = new Composite(parent, SWT.NONE);
    }
    public void setFocus() {

    }
    public static void main(String[] args) {

}
}
```

如果用户在设计器上添加组件，Visual Editor 设计器将会自动生成构建 createPartControl 的代码。

本书的附带光盘上有 VE 的插件，以及 VE 所依赖的 GEF 和 EMF 插件。

附录 E　CVS 版本管理

版本管理系统可以帮助开发人员有效地管理软件资源（源代码文件、配置文件等）的版本问题；也可以帮助开发人员追踪文件的修改记录；防止文件因疏忽而被错误地修改、删除；在小组开发环境中，帮助多个开发人员保持文件的同步；另外，还可以帮助开发人员发现修改过程中产生的 Bug，因此应用 CVS（Concurrent Version System）可以在一定程度上提高软件的开发效率。现在很多开发工具中都集成了 CVS 功能，例如 Eclipse、InteliJ、NetBeans 等；虽然 Eclipse 等集成开发环境（IDE）对 CVS 提供了很好的支持，可以大幅降低 CVS 的使用难度，但是很多开发人员在使用 CVS 时还是不知所措。这是由于用户不了解 CVS 的相关术语及 CVS 的工作模式所致。因此，在附录中将介绍 CVS 的相关术语及 CVS 的工作模式，然后通过一些实例与应用场景，展示如何在 Eclipse 中使用小组开发环境。

另外，CVS 是基于 C/S 模式进行版本管理的系统，本节将介绍如何在 Eclipse 中利用 CVS 进行项目开发，用户可以先安装 CVS 的服务器端程序（CVSNT），然后再通过 Eclipse 进行 CVS 版本管理的操作。

E.1　相 关 术 语

修订版（revision）：CVS 版本管理系统用修订版来管理文件的修改记录，修订版用版本号来表示，即修订版号。对文件的每次修改（提交）都产生一个新的修订版。

资源库（repository）：资源文件的集合，版本管理的容器。在 Eclipse 中被称为 CVS 存储库。

模块（module）：资源文件的组织形式，在版本管理系统中的表现形式为目录（树形结构，可以嵌套）。

输入（import）：将处于资源库之外的软件模块登录到资源库。

输出（export）：从资源库中取出模块。使用 export 方式取出的模块拷贝不包含版本管理的相关信息，对该模块拷贝的修改也不能反映到资源库。

工作拷贝（working copy）：版本管理系统是一个典型的 Client/Server 系统。用户对资源的修改不是直接在 Server 端进行的，而是根据资源库的内容创建一个本地的工作拷贝，用户在工作拷贝中工作，工作完成后再将修改的内容提交到资源库。

签出（checkout）：获得工作拷贝的操作。此前使用过 Visual Source Safe 的读者需要注意，在 Visual Source Safe 中的 checkout 用于锁定文件。

签入/提交（checkin/commit）：将对工作拷贝的修改反映到资源库中的操作。在 CVS 中使用的术语是提交；在 Visual Source Safe 中使用的术语是签入。

更新（update）：将资源库中的最新状态反映到工作拷贝的操作。

冲突（conflict）：在资源库同工作拷贝之间状态不一致的情况下进行签入或更新操作时，

版本管理系统可能会尽量进行合并，如果版本管理系统不能完全处理上述不一致，就称为产生了冲突。

快照（snapshot）：在某一时刻，模块中文件状态（包括文件内容及其版本管理元信息）的静态影像。

标签（tag）：由于 CVS 以文件为版本管理的基本单位，随着开发的进行，对不同的文件的修改次数是不一样的，各个文件的修订版号会因此而变得参差不齐。这不便于模块的管理。为此可以对某个时刻的快照赋予一个标识名称，标识名称就被称为标签。将来通过标签就可以获得模块在该时刻的快照。通过标签所获得的快照是静态的，不能被修改。在 Eclipse 中，标签与版本（Version）是同义词，一般都用于文件集合。需要指出的是：在很多中文资料里，修订版与版本往往不加区分，有时会将单个文件的修订版也称为版本。例如，“A 文件最新版本是 1.3”，这句话中的版本实际上指的是修订版。因此，需要根据上下文来确定版本的意义。

分支（branch）：分支是一种特殊的标签。从分支中签出的资源是可以被修改的。引入分支是为了更好地支持项目的并行开发过程，例如项目开发中，可以有几组人在不同上的分支上工作。

E.2 工作模式

为了解决因多人同时开发而可能产生的冲突问题，版本管理系统有两种常用的工作模式。

（1）锁定—修改—解锁模式

在这种工作模式中，一个开发人员为了能够修改文件，首先必须锁定文件，锁定文件操作赋予了开发人员修改文件的权力。从一个文件被锁定后到其被解锁前，其他的开发人员不能再锁定该文件。这种工作模式适用于小规模的开发小组。如果采用这种工作模式，开发人员应尽量少、尽可能晚地锁定文件并尽可能早解锁文件。模式一是 Visual Source Safe 的默认工作模式。

（2）拷贝—修改—合并模式

在这种工作模式中，每个开发人员都从资源库获得自己的工作拷贝，然后就可以自由地在工作拷贝中继续开发，开发完成后再向资源库提交自己的工作成果。如果在提交时产生了冲突，则必须在解决冲突后才能再提交。模式二比模式一具有更好的并发性，因而也适用于中、大规模的开发小组。模式二是 CVS 所采用的工作模式。

锁定—修改—解锁模式是一种悲观的锁定模式，它假定在开发过程中可能会产生大量的冲突；而拷贝—修改—合并模式则比较乐观，它假定在开发过程中软件的设计及开发任务的分配都比较合理（软件的模块化程度高，开发人员一般各司其职），在开发过程中即使会产生冲突，但产生冲突的机率比较小。为了更好地使用 CVS，用户在使用 CVS 时也应该遵循它的设计前提，努力提高软件的设计水平及项目管理能力，否则将陷入疲于解决提交冲突的尴尬境地。

E.3 模块的输入与签出

要想在 Eclipse 中输入和签出 CVS 资源库，首先要建立 Eclipse 与 CVS 服务器的连接，步骤如下。

（1）打开 CVS Repository 视图（Perspective），可以通过如下菜单操作实现：“Window”→“Open Perspective”→“Other”。在出现的对话框中选择“CVS Repository Exploring”。

（2）在 CVS Repository 视图中单击鼠标右键，选择“New”→“Repository Location”。出现如图 E-1 所示的对话框，在对话框中填入适当的信息即可。例如，根据图中的示范信息，将与我们在前面步骤所安装的 CVSNT 服务器建立连接。需要注意的是：在 Host 输入框中可以输入 CVS 服务器的名字或 IP 地址；在连接类型下拉框中选择 pserver，这是 CVSNT+Eclipse 最简便的配置方式。

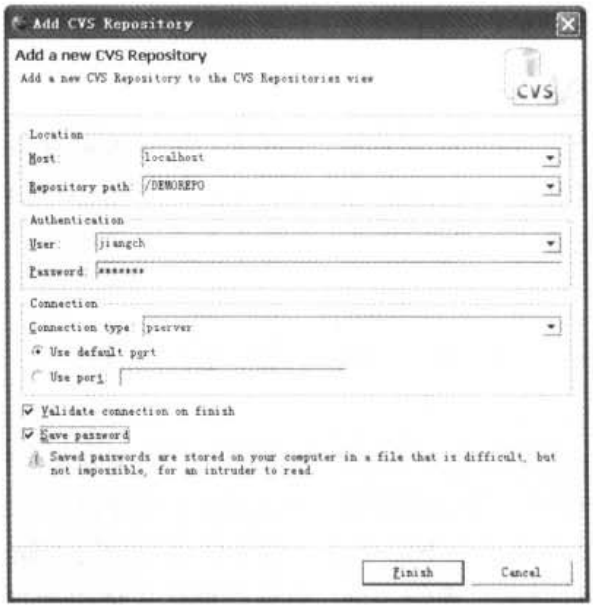


图 E-1　在 Eclipse 中建立与 CVSNT 服务器的连接

在正式的项目开发中，输入模块通常由 CVS 管理人员或项目经理完成，一般的开发人员不需要进行此项工作。在 Eclipse 中，可以将整个项目作为一个模块导入 CVS 资源库。这也是实际工作中经常采用的工作方式。

把项目输入到 CVS 资源库分为两步。

（1）创建要导入资源库的项目。

Eclipse 中的任何项目都可以作为 CVS 资源库的输入，用户可以通过项目创建向导创建作为输入的项目，也可以把之前已有的项目作为资源库的输入。

（2）导入项目到 CVS 资源库。

为了将项目导入 CVS 资源库：打开 Eclipse，选择希望导入 CVS 资源库的项目（project），单击鼠标右键，在出现的弹出菜单中依次选择“Team”→“Share Project”，选择相应的资源库即可按向导把 Eclipse 的项目导入到 CVS 资源库。

Eclipse 导出项目是要把 CVS 资源库中的项目复制到用户的本地，作为 Eclipse 中的项目，用户可以使用此项目进行开发。

如果想获得其他计算机上 CVS 资源库中的项目。可以在 CVS 视图（当 Eclipse 和 CVSNT 建立了连接后，可以通过 CVS 视角找到此视图，如图 E-2 所示）的主分支 HEAD 下的 UnitTest 项目上单击鼠标右键，在出现的弹出菜单中选择“Check Out”，这将签出 CVS 资源库中的模块 UnitTest，并在 Eclipse 建立一个名为 UnitTest 的新 Project。

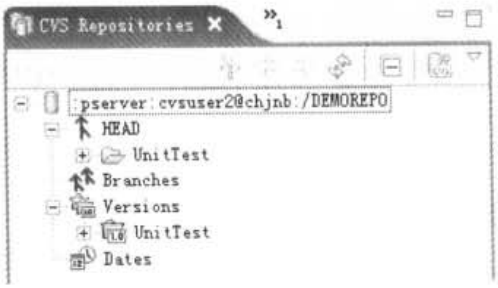


图 E-2　CVS 视图

另外，在菜单中还有一项“Check Out As”，通过此菜单项，可以对签出进行更细致地控制，例如，可以更改 Project 名，可以将签出的内容追加到已有 Project 中。

> **注意**　当签出的为 Eclipse 项目时，Eclipse 会在 Workspace 中利用签出的 Eclipse 项目，但当项目不为 Eclipse 项目时，用户可以选择使用 Eclipse 项目创建向导创建项目，把签出的目录作为源文件目录存在即可。

E.4　日 常 操 作

在具有版本管理的小组开发环境中，有些操作是开发人员几乎每天都要进行的，如提交修改工作、更新本地工作拷贝、解决冲突等。在 Eclipse 中进行这些操作时，如果是对目录进行操作，则 Eclipse 会自动对该目录下的子目录和文件进行相同的操作，对目录的操作是递归嵌套的。在下面的讲解中只以文件操作为例，但这些操作也适用于目录。一般来说，如果在某个目录下要对多个文件进行相同的操作，则选择目录操作比较简便。

在日常操作中，开发人员通常要进行的操作有提交、更新和比较等，这些操作可以通过文件的右键菜单“Team”下的子菜单执行。下面主要介绍这几个操作。

（1）提交（Commit）

因为 Eclipse 的本地版本和 CVS 资源库文件的版本可能会存在差异，下面分四种不同的工作场景介绍如何将工作拷贝的修改提交到 CVS 资源库。

第一，签出模块后，在本地对文件（即 Eclipse 项目中的文件）进行了修改，CVS 资源库中的文件没有变化。

在工作拷贝中对 SimpleTest.java 做一些改动并保存，此时 Eclipse 中项目的画面发生了改变，见图 E-3。在改动的文件及其所属的目录前都出现一个箭头，在希望提交的文件上单击鼠标右键，在出现的弹出菜单中选择“Team”→“Commit”，在随后出现的对话框中输入注释后单击“OK”按钮。虽然注释不是必须输入的，但是，最好养成在每次提交时写注释的良好习惯，这在实际的系统开发过程中是很有必要的。提交成功后文件的修订版本号会自动增加，SimpleTest.java 的修订版号从 1.1 变成了 1.2。

第二，签出模块后，在项目中新建了新文件，希望将新建的文件追加到 CVS 资源库中。

在新建的文件上单击鼠标右键，在出现的弹出菜单中选择“Team”→“Add to Version Control”。需要注意的是，上述操作并没有把新建的文件追加到 CVS 资源库中，为了把新建的文件追加到 CVS 资源库中，还要执行下列操作：在新建的文件上单击鼠标右键，在出现的弹出菜单中选择“Team”→“Commit”。

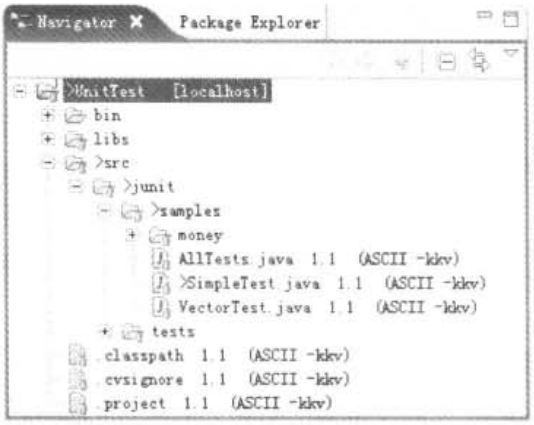


图 E-3　CVS 签出的文件被修改

第三，签出模块后，在项目中删除了文件，希望在 CVS 资源库中也删除相应的文件。

首先在项目中删除文件，然后在被删除文件的父目录上单击鼠标右键，在出现的弹出菜单中选择“Team”→“Commit”。值得指出的是：上述操作只是将被删除的文件从主分支上删除了，但并没有从 CVS 资源库中完全删除，也不应该从 CVS 资源库中被完全删除，因为某个标签或分支可能需要被删除的文件。

如果想恢复被删除的文件，在 Eclipse 的 Navigator 视图中选中被删除的文件的父目录，在右键菜单中选择“Team”→“Restore from Repository”，然后在出现的对话框中选择希望从哪个修订版恢复被删除文件的内容。

第四，签出模块后，在本地对文件进行了修改，但在提交之前，其他的开发人员已经对同一个文件进行了修改并提交到 CVS 资源库中。

此时，由于 CVS 资源库中该文件的修订版本号将高于本地该文件的修订版本号，因此将不能提交该文件。如果试图提交该文件，Eclipse 将弹出一个错误对话框（但此操作不会覆盖 CVS 中的文件，也不会修改本地文件），见图 E-4。在这种情况下，为了能提交修改必须首先解决冲突。

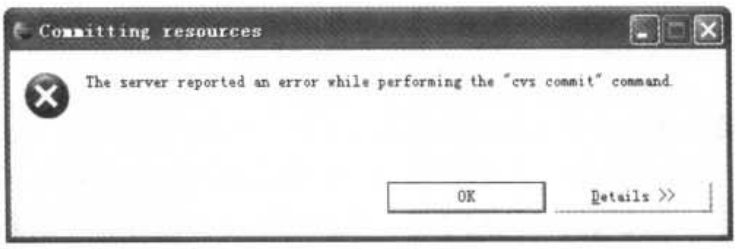


图 E-4　版本冲突对话框

在小组开发环境中，应该经常性地提交自己的工作成果，但也不能随意提交。一般来说，以提交的更改不对其他开发人员造成妨碍为前提条件。例如，提交的代码应能够被正常编译。如果在小组开发环境中还使用了诸如 CheckStyle 之类的代码审查工具，则提交的代码除了能够被正常编译之外，还应能通过代码审查工具的审查，否则整个团队都可能会因为提交的错误代码而受影响。

（2）更新（Update）

在系统的开发过程中，开发人员不应只关注自己的开发，还应尽量与其他开发人员保持同步，也就是需要经常地将其他开发人员的开发成果更新到本地的工作拷贝中。更新操作有如

下几种场景。

第一，CVS 资源库中的文件被更新了，但本地文件没有更新。

在开发小组中，可能有部分开发人员负责一些通用模块的开发，而其余的开发人员则专注于业务系统的开发。在这样的开发场景中，业务系统的开发人员就应经常更新通用模块，保持工作拷贝中的通用模块与 CVS 资源库中通用模块的同步。在文件上单击鼠标右键，在出现的弹出菜单中选择“Team”→“Update”。本地文件的内容将被替换为 CVS 资源库中相应文件的内容。

第二，CVS 资源库中的文件没有被更新，但本地文件被更新了。

此时，由于本地文件的修订版号与 CVS 资源库中文件的修订版号相同，更新操作实际上什么也不做。

第三，CVS 资源库中的文件和本地文件都被更新了。

此时，Eclipse 会尽量将 CVS 资源库中的内容合并到本地文件中。如果在合并过程中没有发现冲突，则合并后的文件将包含本地修改和其他开发人员的修改。如果在合并过程中产生了冲突，则 Eclipse 会在合并文件中着重显示冲突的部分，如图 E-5 所示，并将原来的本地文件做一个备份。用户应该将产生冲突的部分修改为正确的代码，解决版本冲突的问题后就可以提交新版本了。

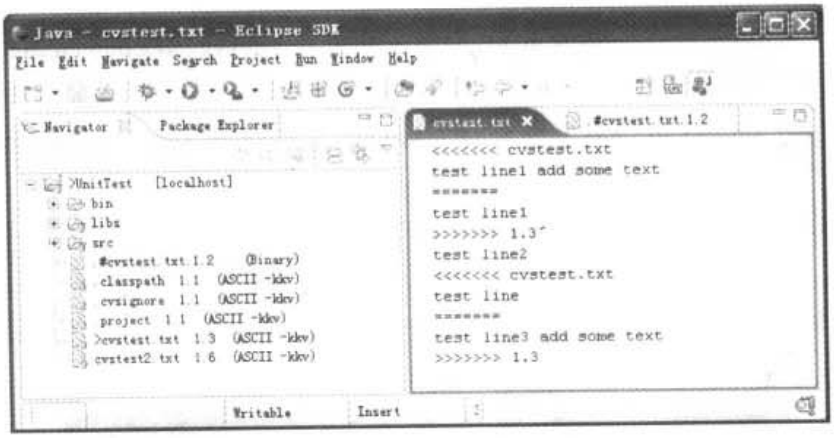


图 E-5　版本冲突

注意　如果修改后用户本地的版本和 CVS 的版本产生冲突，Eclipse 会把两个版本合并，并用“<<<<...”和“>>>>...”标出两个版本之间的不同。在合并版本时，不要把其他人的修改删除掉，这是新手在开发时有可能会犯的错误，而且这是非常严重的错误。如果不敢确信自己的修改，最好能找修改了此文件的作者一起合并版本。

（3）比较、替换及修改记录

在软件项目的开发进程中，可能需要参考历史版本的信息，甚至将某个文件的内容回滚到某个历史修订版。在 Eclipse 中也提供了这方面的功能。文件或目录的右键菜单“Compare With”提供了和其他修订版相比较的功能；右键菜单“Replace With”提供了将工作拷贝的内容替换为其他修订版内容的功能。例如，用户修改了工作拷贝的某个文件后又想放弃修改内容，此时，可以选中文件并在右键菜单中选择“Replace With”→“Last from Head”（假定 CVS 资源库没有被其他人员更新过，否则选择“Replace With”→“Revision”）。上述操作对目录而言，

实际上是对该目录中的所有文件执行操作。

在 Eclipse 中，通过文件的右键菜单“Team”→“Show Resource History”可以方便地查询文件的修改记录，例如文件 cvstest.txt 在某个时刻查询到的修改记录，见图 E-6。从中可以看出谁在什么时候修改了文件的内容，也可以看出在每次提交时都写注释会有助于追踪代码的修改意图及今后的代码维护工作。

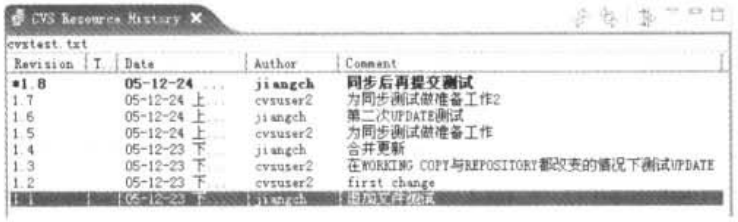


图 E-6　文件的修改记录

经验而言，用户在每一次修改之前都应该更新当前文件的内容，使之和 CVS 资源库同步，从而尽量避免产生版本冲突，每一次提交时，都应该保证把修改的文件提交到 CVS 资源库，至少要保证不能因为此次的提交影响其他人的开发。